CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Multi-level FaaS Application Deployment Optimization

Master's thesis in Computer science and engineering

Junpeng Zhang

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Multi-level FaaS Application Deployment Optimization

Junpeng Zhang

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Multi-level FaaS Application Deployment Optimization

Junpeng Zhang

Supervisor: Joel Scheuner, Department of Computer Science and Engineering
Examiner: Robert Feldt, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Multi-level FaaS Application Deployment Optimization

Junpeng Zhang
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Functions as a Service (FaaS) has become a trend in software engineering due to its simplicity, elasticity, and cost-effectiveness. FaaS has drawn both the industry's and researchers'/practitioners' attention. We notice that more applications are shifted to cloud platforms; however few studies are conducted on how to deploy a FaaS application in a cost-efficiency way. From the perspective of deploying a FaaS application, resource allocation optimization and application-level latency reduction are the two factors that affect the overall performance and total running cost of a FaaS application. Currently, many developers manually analyze the execution logs and run multiple trials to predict a proper deployment strategy or just deploy functions with the finest granularity by default. Such tasks require a considerable amount of human effort, and it has to be done repeatedly whenever the FaaS platform carries out performance-related upgrading. To mitigate this problem, we explore several potential solutions and implement a highly automated framework, which can optimize the deployment of an application from both the perspectives of memory allocation and application-level latency reduction. This study has been conducted by following the guideline of design science research methodology. Afterwards, a controlled experiment is performed to evaluate the framework. The preliminary evaluation reveals that the framework successfully delivers the optimal strategies for cheapest, fastest, and trade-off balanced (on the specific test case, the framework identifies a 10.5% speed gap and 13.3% cost difference between the most optimal case and the worst case). Furthermore, the framework is open-sourced on GitHub for further studies.

Keywords: Function as a Service, FaaS, deployment optimization, memory allocation, fusion, latency reduction.

Acknowledgements

I want to offer special thanks to my academic supervisor, Joel Scheuner, for his continuous guidance and overall support during my master thesis. My sincere thanks go to my examiner, Prof. Robert Feldt, for the valuable feedback. I also would like to thank Jun-Ze Lai for valuable insights of his thesis work, and Victor Jarlow for the valuable feedback at the academic writing seminar and being the final presentation opponent. Last but not least, my personal thanks go to my family for supporting me to come back to school that I dropped out nine years ago and finish the thesis work. Without the help and support from any of you, I won't be able to finish my thesis work. Thank you all!

Junpeng Zhang, Luxembourg, November 2021

Contents

List of Figures

List of Tables

1
Introduction

Cloud computing has become more and more popular during the last decade. Cloud providers, such as Amazon's AWS [1], Microsoft's Azure [2], Google Cloud [3]., are offering different solutions to meet customers' needs. Such solutions can be categorized by the type of services, namely infrastructures (IaaS, Infrastructure as a Service), platforms (PaaS, Platform as a Service), software (SaaS, software as a Service), and Functions (FaaS, Function as a Service). IaaS provides low-level computing resources including hardware, operating system and networks in a virtualization way (i.e. virtual machines). As the evolution from IaaS, PaaS offers a development platform that allows users to focus on developing software applications without worrying about infrastructures, especially when managing and scaling servers. FaaS, similar to PaaS, delivers a way of developing function-level microservices and more cost-effective practice of the pay-as-you-go principle of serverless computing than PaaS.

Due to the trend of shifting from code-heavy monolithic applications to smaller, self-contained microservices in the industry [4], it is not a coincidence that serverless computing has been introduced and grown fast recently. Castro et al. [5] defined serverless computing as "a computing platform that hides server usage from developers and runs code on-demand automatically scaled and billed only when the code is running." In contrast to monolithic systems, serverless computing allows developers to decompose products into building blocks that do only one function and do it well [6]. FaaS, as a modern type of serverless, focuses on function-level development with more outstanding advantages: great simplicity, unlimited elasticity, and ideally cost-efficiency.

Simplicity is an essential advantage. Developers do not have to do anything to cope with the scalability, which the platform takes care of. This kind of ease-to-use also applies to other operational concerts, such as deployment and monitoring. Developers just need to focus on functionality development. The study [7] reports that 34% of the applications in their dataset choose serverless because of simplicity. Another advantage is elasticity, which is not saying that traditional applications are not scalable, but it requires much more technical effort than serverless applications. FaaS provides almost infinite scalability with much less technical effort. For example, with AWS Lambda (FaaS provided by Amazon) [8], the platform automatically offers up to 3000 cumulative instances to cope with the initial burst and 500 more per minute later on. Study [7] shows that elasticity is an important reason that 34% of their observed applications choose serverless. One key advantage of serverless is

the model of pay-as-you-go. Study [7] found a substantial amount of idle resources (e.g. CPU and memory) on the VMs that many cloud users allocated and paid for. On the other hand, by adopting FaaS, such idle resources can be avoided, and billing can be scaled to zero. What is better, FaaS as a type of serverless computing than other cloud services, e.g., IaaS and PaaS, has a substantially lower granularity of billing [5]. With IaaS, users have to pay for their virtual machines at least per minute, even hours and years if users want a discount price. In contrast, with FaaS, users are only asked to pay as low as one millisecond of function execution time.

Because of these advantages, more and more users have started to practice FaaS to benefit from such flexibility. According to a report published in February 2020 [9], almost 50% of AWS users and nearly 80% of AWS container users have adopted AWS Lambda. Such a noticeable transition draws great attention from practitioners. Much research on this topic has been done to help understand the concept and conduct better practices. Of which, one direction about orchestrating multiple single deployed functions in order to complete more complex business has become heat.

When starting a new development of microservices-based systems or moving a monolithic system to a microservices-based FaaS platform, in order to benefit the function-level complexity and achieve cost-efficient practice, developers need to take into account not only how to design each function by following the paradigm of "loose coupling high cohesion", but also how to orchestrate multiple functions if needed with a cheap deployment setting which makes sure it runs for a short duration with enough computing resource (memory, CPU etc.).

Figure 1.1 gives an example of FaaS application. It contains both sequential (e.g. Step 1 -> Step 2 -> Step 3) and parallel workflows (e.g. Step 1->Step 2->Step 3, meanwhile Step 1 -> Step 4). Input data will be passed into the root function “Step 1” for pre-processing; then functions “Step 2” and “Step 4” will take part of the processed data from “Step 1” and continue the workflow in parallel. Function “Step 4” will end when it finishes its job; meanwhile, function “Step 2” will invoke function “Step 3” in sequence and ends when “Step 3” finishes its job. This example will be used through this study to illustrate how to optimize the deployment of such a FaaS application dynamically.

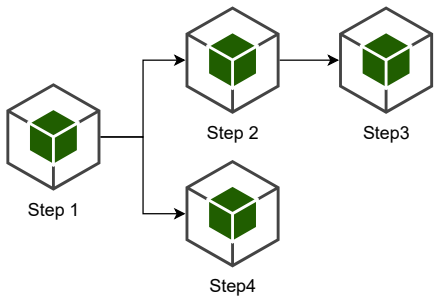


Figure 1.1: Example of FaaS application, where functions have been orchestrated in chain and in parallel.

The most common way of deploying such a FaaS application on the FaaS platform will be like Figure 1.2. The application is deployed in the finest granularity, i.e., each function is deployed by itself with specific resource allocation settings. However, from the perspective of the pricing model of the public FaaS platform (resource and duration), such a deployment strategy may not be the most optimal solution. A possible better deployment strategy may be combining some functions into a deployment unit to reduce loading up duration, i.e., application-level latency (cold start). For example, as Figure 1.3 shows, if Step 1, Step 2, and Step 4 share some same dependencies, e.g., external libraries. etc., fusing them with a proper resource allocation might reduce total billed duration, and eventually improve the overall performance and reduce the total cost.

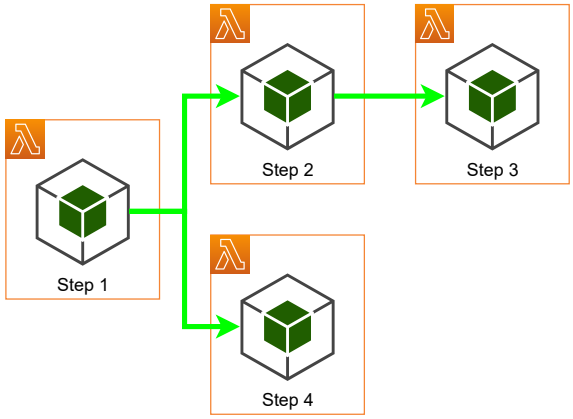


Figure 1.2: A common way of deployment the given FaaS application example.

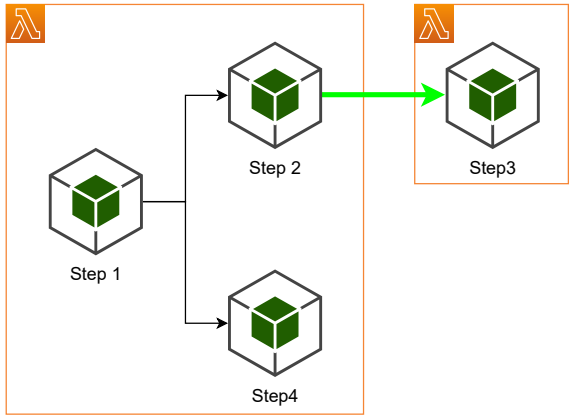


Figure 1.3: A possible optimal deployment strategy for given FaaS application example.

1.1 Research Questions

This study aims to design and implement a multi-level (resource allocation and application-level latency reduction) optimization framework for deploying a FaaS application with little human effort in practice. Such a framework should perform

resource allocation tasks and fusion tasks on a given FaaS application and come out with an optimized deployment strategy at the end. The strategy should shorten the running time by orchestrating related functions at the deployment phase and assign an optimized amount of memory for the new fused deployment. Afterward, a standalone controlled experiment will be performed on the framework to quantify the improvements in latency reduction and the cost benefits. Hence, the following Research Questions are addressed:

RQ1: What strategies should be applied to design and implement an automatic memory allocation module?

This question intends to seek a suitable strategy to perform memory allocation tasks. Most studies nowadays focus on optimizing memory allocation for a single FaaS function, but few studies have been conducted for application, i.e., a set of orchestrated FaaS functions. Are the existing strategies as good for optimizing orchestrated FaaS functions as for a single FaaS function? If not, is it enough to modify the existing strategies to cope with orchestrated functions?

RQ2: How can an automatic memory allocation module and a latency-reducing focused fusion module in the same framework optimize the deployment of a FaaS application at the same time?

This question aims to find a proper way to combine the resource allocation module and a latency-reducing fusion module. A synchronized combination or a blended mixture should be discussed and examined to make the new extended framework work productively and efficiently.

RQ3: How effectively does the new multi-level FaaS application deployment optimization framework optimize the deployment?

This question seeks to quantify the evaluation results of the new framework. A standalone evaluation should be carried out to check if the framework works as expected in different criteria.

1.2 Thesis Outline

The remainder of the rest of the thesis is structured as follows: **Chapter 2** introduces the background of this study in terms of resource allocation and latency reduction. **Chapter 3** presents related work in the areas of memory allocation strategy, application-level orchestration and benchmark. **Chapter 4** describes the research methodology this study follows. **Chapter 5** presents the design and implementation of the new framework by explaining how each module is designed to solve the task and how all modules work together as a standalone framework. **Chapter 6** presents and discusses the evaluation results. Finally, **Chapter 7** summarizes the contributions, concludes this thesis and outlines future work.

2
Background

This section introduces the necessary background for the understanding of this thesis in the areas of memory allocation, latency reduction, and fusion orchestrating and fusing.

The trickiest challenge comes from the process of achieving cost-efficiency. Nowadays, most public FaaS providers introduce similar pricing policies that commonly consist of a fixed and dynamic part. The fixed part is charged generally based on the number of invocation requests, whereas the dynamic part is charged on the combination of execution duration and allocated resource. To achieve an optimal cost-efficiency result, besides the quality of code deployed, the FaaS developers are required to make a smart decision on deployment configuration in terms of memory allocation and reducing latency on application-level at the deployment phase.

2.1 Memory Allocation

Though FaaS will do many server configurations without human interaction, it leaves the tricky one for the developers, memory allocation. Memory allocation in FaaS is not only about assigning memory but also comes with computing power (i.e. CPU). Until now, more computing power still only comes with larger memory allocation on most public FaaS providers, e.g. Amazon AWS and Google Cloud. For example, running CPU intensive tasks (e.g. video processing), more memory needs to be assigned to obtain more computation resources. More resources (memory and CPU) might make functions run faster and therefore shorten the usage period. However, when it comes to the bill, charges of operation period and allocated resource will always go against each other. To find a balance point on a charge-memory related curve, the best practice is to observe the log and perform the memory allocation based on prediction. Although most public FaaS platforms offer tools to track and log the usages closely, developers still need to analyse the log and adjust the memory allocation manually.

Several strategies have been brought up to address the memory allocation problem. A feasible approach is the exhaustive searching method. It first tests a couple of deployment settings with different memory allocation settings (e.g. on AWS Lambda, test with 128MB, 256MB, 512MB, 1024MB ... up to 10GB). Then, either a cost-effective or trade-off solution will be drawn based on runtime feedback. For example, AWS Lambda Power Tuning [10] has adapted this strategy. Others adopt more com-

plicated strategies to predict an optimal memory allocation. E.g. COSE [11] uses statistical method Bayesian Optimization, and Sizeless [12] uses a pre-trained model.

However, all these tools only focus on optimizing memory allocation to reduce the cost. Another significant factor shall not be ignored: latency reduction.

2.2 Latency Reduction

To shorten the execution duration, besides increasing the quality of the source code deployed on the FaaS platform, reducing the latency caused by the FaaS platform is important. Czentye et al. [13] pointed out that end-to-end latency might occur because of the internal operations, involved techniques, and the platform's available configuration.

One type of latency, for a single function, is cold start. Unlike warm start (functions already in memory), cold start happens when a function is triggered the first time by an event (e.g. HTTP request, scheduled timer, etc.). It takes time for the FaaS platform to create a container and initialize the function. The function will stay alive for a short while (minutes at most, depending on the FaaS platform) after executing the previous request. The subsequent coming request will be handled faster since the function is already loaded in the memory and ready to go (warm start).

Furthermore, when a function is triggered by multiple parallel requests, the FaaS platform automatically scales by initializing multiple VMs with containers of the function to handle the requests. Each of these scalings brings up latency. One feasible way is reducing the number of "cold start" by adopting different strategies. E.g. Shahard et al. [14] presented a solution that can pre-warm the functions by predicting when the subsequent invocation request arrives.

Another type of latency occurs when multiple functions are required sequentially to complete a more complex request. In this scenario, the platform may spin up multiple VMs which are not even on the same physical server. The communication between functions will take time. Since FaaS is stateless, passing data at runtime is impossible. To make a group of stateless functions working together depends on stateful storage (e.g. database). Such latency occurs not only when initializing each function (cold start), but also at the phases of storing and fetching data from the database and network delay between physical servers. For instance, a case study [15] reported that a task of signing up Autodesk's account took ten minutes on average due to the overheads of how FaaS platforms conduct task handling and state management.

Scheuner and Leitner [16] proposed a method that groups several functions together by transpiling the source code into one function. Another well-known study in this area conducted by Czentye et al. proposed a solution to optimize latency-sensitive applications on AWS by optimizing the software's layout (i.e. "calculate optimal groupings, selects flavors, triggers and state storage") [13]. Lai recently introduced

a fusion framework optimizing the deployment optimization by fusion related functions at the deployment phase based on runtime feedback data [17]. The fusion framework does not affect the code level, and it is for general purpose without specific latency prerequisite. However, Lai's fusion framework is missing the functionality of automatic memory allocation. Lai argued that extending a memory allocation module to the fusion framework will increase the complexity and bring heavy overhead.

Therefore, this study intends to explore an optimization strategy for general purpose with the ability to optimise resource allocation and reduce latency at the same time.

2.3 Functions Orchestrating and Fusing

When deploying a FaaS application on a cloud platform, a common way is to deploy each function into a separate container. Then, according to the workflows, either using a standard orchestrator tool provided by cloud platform or a built-in invoke method to orchestrate each function into an application. Taking AWS as an example cloud platform, it provides AWS Step Functions as a "serverless function orchestrator" [18], as well as a built-in Lambda invoke function for invoking other serverless services.

AWS Step Functions helps to chain multiple Lambda functions into "business-critical" applications [19]. Besides sequencing, it also provides possibilities of retry, map-iterate, error handling, and debugging features. Each step is logged and can be easily traced. All of these features enable developers to easily and quickly develop and deploy applications. However, one drawback is that until today the cost of applying AWS Step Functions is rather expensive. Besides the running cost of each Lambda function in the chain, AWS charges users the cost of state transitions, e.g. US$ 0.025 per 1000 state transitions of standard workflows. Such cost is some extra expense compared to deploying an application without using AWS Step Functions. Applying AWS Step Functions has no impact on reducing latency; it just helps to link all functions as defined.

AWS Lambda built-in invoke function gives the possibility to developers to dynamically invoke other resources as they wish. It will not generate operating costs like using AWS Step Functions, but requires extra work to handle retry, retry, map-iterate, error handling, debugging and logging programmatically by the developers. As same as AWS Step Functions, it also has no impact on reducing latency.

In order to reduce application-level latency, developers have to consider adapting function fusing methods, i.e. locally invoke subsequent functions. For example, locally invoke a function by using the "require" function in Node.js or "import" function in Python. Locally invocation will reduce the latency and benefit both cost and performance. However, whether to invoke functions locally and remotely requires extra efforts from developers to find out. Given the FaaS application example in Section 1., an application with four functions will have 15 ways of deployment

possibilities (illustrated in Appendix 9.1), and an application with five functions will generate 52 ways of deployment possibilities (Bell Numbers [20]). Although it is not necessary to test all deployment possibilities, some of which can be ruled out by applying pre-defined constraints (e.g. it is not cost-effective to fuse a high resource required but low execution duration function with a low resource required but long execution duration function together), it still brings a significant amount of human effort to test each deployment strategy. This study aims to create an automation framework that helps to search for the optimal deployment strategy for a FaaS application.

3
Related Work

Some research has been done to explore possible ways to optimize the function's deployment built on FaaS. Most focused on memory allocation optimization on a single function, but holistic deployment optimization drew little attention. This chapter presents related works in optimizing memory allocation, application deployment optimization, and FaaS platform benchmarking.

3.1 Memory Allocation Optimization

Even though FaaS platform providers simplified most of the provisioning requirements, memory allocation is still a tricky task left for the users. The users must manually configure how much memory and computer power their deployed application requires based on their own experiences. Zhang et al. [21] conclude that memory configuration to optimize for cost and performance is a non-trivial task.

A set of tools are available to help to predict an optimal deployment configuration. Nowadays, The most popular one is AWS Lambda Power Tuning introduced by Casalboni [10], a popular open-source tool on GitHub [22] with more than 2000 stars. The tool will run a serverless function with a predefined subset of memory configurations and output the execution time and cost with each memory setting. Based on such results, users can make appropriate memory settings to achieve optimal deployment. This study has adapted this strategy as it has been widely used and proved in the community. However, the AWS Lambda Power Tuning is designed for a single Lambda function; we restructured the tool by using AWS Step Functions to make it suitable for application-level, i.e., performing memory optimization tasks on multiple functions simultaneously. Furthermore, we extend the usage of the output of the AWS Lambda Power Tuning tool by combining and passing to our fusion module in the framework, where the data will be used to perform further fusion simulation tasks.

Another approach is COSE, a framework that uses Bayesian Optimization model to statistically learn the relationship between cost/runtime and unseen configurations of a serverless function [11]. The framework supports function chaining and can adapt to changes in the execution time of a serverless function. We see COSE as an improved exhaustive searching method (e.g., AWS Lambda Power Tuning). It still needs to run multiple tests with different configurations first, and then it applies a statistical learning approach to the testing results to predict an optimized configu-

ration. We believe this approach can effectively generate more precise predictions. Especially after AWS changed their policy of Lambda memory allocation on Dec. 1st 2020, from 64MB increments to 1MB increments and 3GB max limit to 10GB max limit [23], it will be more costly to run exhaustive search methods. However, like AWS Lambda Power Tuning, it focuses on memory size optimization of serverless functions but not on latency reduction. Also, due to the lack of implementation details and insufficient evaluation results, we decided not to adopt this approach for this study. Even though we didn't select it as part of our solution, we believe using statistic method to predict memory size for a FaaS function is feasible and maybe more efficient. Hence, we designed our framework following the "pipeline" model and leave the possibility to easily replace the memory allocation module with other solutions, e.g., a statistical search module like COSE in the future.

Sizeless is an approach that predicts the optimal resource size of a serverless function using monitoring data from a single memory size [12]. First, a multi-target regression model is trained based on a dataset of time/memory relationship generated by running synthetic functions with various memory settings. The model will then predict how a serverless function in a realistic setting behaves for all memory sizes based on monitoring data for one single memory size. Sizeless offered a trade-off factor to optimize either by lowest cost or balanced, which could be another possible strategy applied to this study. Different from COSE and AWS Lambda Power Tuning those have to measure multiple memory sizes to predict, Sizeless adapts a large pre-trained model to predict with just one measurement. However, we have concerns about how well such a pre-trained model reflects on the frequently upgrading FaaS environments. Major upgrades in infrastructures, runtime environments, etc., are believed to have dramatic impact on the prediction of the previously trained model, such a prediction might not be suitable for the new environment.

3.2 Application Deployment Optimization

Few studies focus on optimizing FaaS applications on the layout level. Scheuner and Leitner [16] envisioned a method that groups several functions together by transpiling the source code into one function. Such a code-level optimization is out of the scope of this study.

Other studies like Czentye et al. [13] proposed a strategy to optimize the application layout based on dynamic performance measurements. They first collected performance data of the selected cloud platform by performing synthetic microbenchmarks. With the latency requirements as input, their algorithm will redeploy the application by changing the system layout (i.e., calculating optimal groupings, selecting flavours, triggers, and state storage) in a cost-effective manner. Our framework also changes the deployment layout by fusing multiple functions together to reduce application-level latency. However, our framework is designed for general purposes without the user's input of latency requirements. We extract the execution data by running black-box testing instead of using performance indicators of the FaaS

platform that collected by performing micro-benchmarks beforehand. During the relatively short period of our thesis work, we have noticed several upgrading of the FaaS platform in terms of pricing model and performance improvement (software and hardware). Such a dynamic environment requires swift update of previous built model, which involves significant workload. For a general purpose usage, it is apparently over-killed. Moreover, our framework also returns a trade-off balanced strategy other than simply the fastest or cheapest solutions.

Another tool, AWS Lambda Fusion framework introduced by Lai [17], can automatically reduce latency by improving deployment strategy. The fusion framework continuously applies two algorithms (Hill-climbing and Heuristic) to find an optimal deployment configuration (how to deploy the functions, separately or in groups) based on runtime feedback data. However, the fusion framework lacks a memory allocation module and a controllable mechanism of the fusion process's performance (i.e., how long the fusion framework takes to optimize a specific application). The framework implemented in this study was inspired by the AWS Lambda Fusion framework but with multiple significant modifications and improvements, including adding a resource allocation module and a controllable mechanism.

3.3 FaaS Platform Benchmark

Various benchmark types have been commonly conducted to help FaaS users quantify the performance-related challenges on different platforms. Scheuner and Leitner [24] categorized FaaS performance benchmarking into two types: micro-level benchmarks and application-level benchmarks. The former type commonly uses artificial workloads to measure the performance of specific aspects (e.g. floating-point CPU computer power), whereas the latter type focuses on measuring the end-to-end response time of a realistic application.

While some studies focus on designing specific benchmarks to examine their specific targets, others introduce general-purpose benchmark suites. FunctionBench [25], a cross-platform suite, is composed of microbenchmark and application workload. Among different types of workloads offered, Image Processing workload works on measuring CPU and memory performance exclusively. PanOpticon, introduced by Somu et al. [26], provides the option to measure the effects of features such as function chaining and choice of function triggers instead of focusing on tuning resource parameters like memory, CPU requirements and measure metrics like execution time. Another benchmark suite FaaSDom [27], provides cross-platform benchmark tests written in four programming languages (i.e. Node.js, Python, Go and .Net Core). It is also able to estimate budget costs by an integrated model.

Although many of the benchmark suites are open-sourced, e.g. FunctionBench [25] is available on GitHub [22] and a fork of FaaSDom [27] (original project not available), others remain closed sources. In Scheuner and Leitner's literature review [24], a key finding addressed that many studies lack reproducibility, which they believed is particularly important for both types of benchmarking. Therefore, it is decided

to release this study's outcomes to the open-source community by the end for future study.

Because this study focuses on developing a prototype, a comprehensive set of benchmark test cases is out of the scope. To evaluate the framework within the time constraints, we decide to adapt some previously mentioned benchmark tools to simulate our tests. The test case used in this study consists of resource-sensitive functions, time-consuming functions, and other simple functions. The workflow of the test case will be a combination of chain and parallel. Hence, it is believed to be a good start for this thesis work.

4

Research Methodology

This thesis aims to create an that shall help resolve a practical problem when optimizing the deployment of a FaaS application. Design science research methodology is considered a practical guide to follow for this thesis work. The rest of this section is constructed as follows. First, we briefly introduce the design science research method. Then, we present the three iterations that have been conducted through this thesis work. Lastly, we describe how we evaluate the after the three iterations.

4.1 Design science research

Design science research [28] was generalized by Wieringa to solve practical problems and is usually conducted in a regulative cycle shown in Figure 4.1. As elaborated by Wieringa, the regulative cycle typically starts with 1) problem investigation, in which phase the researcher shall aim at identifying and describing the problem; based on the understanding of the problem, the researcher shall 2) design solutions which supposed to be able to cope with the addressed problem or reduce the gap in-between; then, 3) such solutions shall be validated if they would indeed server the purpose of solving the problem; after a possible solution is validated, the researcher shall 4) implement such a design accordingly, and at last 5) evaluate if the implementation meets the design requirements.[28] Such a regulative cycle will be iterated multiple times to gradually reduce the gaps between the problem and expectation and eventually solve the problem.

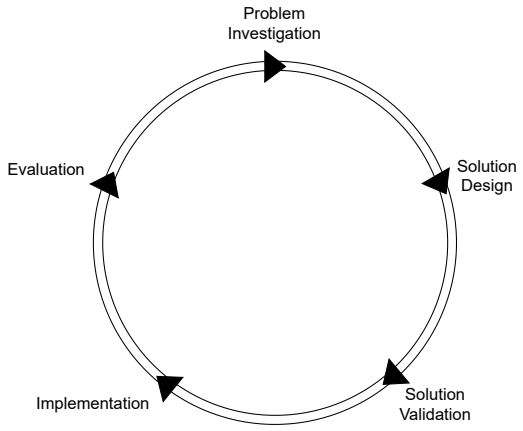


Figure 4.1: Design science research regulative cycle.

4.2 Iterations

In this thesis work, we first carried out three iterations to design and implement the framework. Table 4.1 lists the activities that have been conducted in different phases of each iteration.

	Iteration 1	Iteration 2	Iteration 3
Problems	How to optimize the deployment of a FaaS application?	Artifact worked on the simplest test case. Try on complex test cases, and adjust the algorithms if needed.	Throttling exception for invoking AWS SDK too frequently. Reduce workload if possible.
Focus	1. Design of the artifact. 2. Test case development. 3. Implement the artifact if possible.	1. Increase test case complexity 2. With the new test case, to improve algorithms of the artifact.	Continuously to improve on the artifact from both aspects of functionality and performance.
Problem Investigation	1. Literature review; 2. Discussions with domain expert / academic supervisor - J. Scheuner.	1. Result analysis of iteration 1; 2. Discussions with domain expert / academic supervisor - J. Scheuner.	1. Result analysis of Iteration 2; 2. AWS Lambda Throttle Exception investigation.
Solution Design	1. Memory allocation module: - AWS Lambda Power Tuning - Prediction based on trained model - Bayesian Optimization 2. Combine two modules: - Step Functions 3. Other design decisions: - Dynamic vs Local mocks 4. Test case design	Algorithm adjustments - No Jump back refinement - No double entry refinement - Memory allocation from 1.3x to 1.0x	New Functionality - Based on the outcome of AWS Power Tuning, simulate the running cost of each strategy before actually executing it. - “Balanced” strategy
Validation	1. Weekly discussion with domain expert/academic supervisor; 2. Self examine	1. Weekly discussion with domain expert/academic supervisor; 2. Self examine	Self examine
Implementation	1. Artifact architecture 2. Artifact v 0.1 3. Test Case 1	1. Artifact v 0.2 2. Test Case 2 & 3	Artifact v 0.3
Evaluation	Test case 1	Test case 2 & 3	Test case 2

Table 4.1: Activities of Each Iteration

4.2.1 Iteration One

The initial iteration weighs more than the other two in this study. During this

iteration, there was a heavy focus on designing the fundamental architecture, implementing the framework, and developing the test case to represent the real-world applications as much as possible. By the end of this iteration, we managed to deliver a high-level architecture of the framework, an initial version of the framework, and a set of test cases. We explored and discussed several possible alternatives during the design phase and came out with the current design. A comparison of these alternatives is discussed in Section 5.1. Furthermore, with the help of several studies in the field, we categorized the real-world FaaS application into four types from the perspective of the pricing model of the FaaS platform (i.e., resource required and execution duration), and we created two functions and mixed them to simulate these four types. A detailed discussion about the test case is presented in Section 5.5.

4.2.2 Iteration Two

The focus shifted toward algorithm refinement during the second iteration. Based on the evaluation results from the previous iteration and discussion with the domain expert / academic supervisor, we decided to increase the complexity of the test cases to simulate more complex scenarios. We eventually applied several adjustments to the algorithm of the framework. All the algorithms help to generate the deployment strategies are stated Table 5.1.

4.2.3 Iteration Three

When evaluating iteration 2, we encountered the throttling problem raised by the AWS Lambda platform. We realized that the framework invokes AWS SDK too frequently. We aimed to solve this problem by reducing payload and improving the framework's performance during this iteration. Eventually, we successfully extended the framework with a new function, which simulates the running cost of each strategy based on the outcome of AWS Lambda Power Tuning before actually executing it. With the same test case of iteration 2, the local simulation function eliminated half of the strategy candidates before actually testing it in the cloud. Such an improvement has improved the stability of the framework by reducing invocations of AWS SDK, consequently reduced execution cost for less invocation of Lambda functions. However, we could not implement similar functions that can simulate the execution elapsed time and trade-off balanced value during this iteration due to time constraints. Furthermore, we extend the analyzer module (Section 5.3.4) with a new feature to calculate and select a trade-off balanced strategy based on execution data and the balanced weight parameter.

4.3 Evaluation of the framework

After three iterations, we carried out a standalone evaluation of the framework by using a new set test case (described in Table 6.1). We perform twenty times of deployment optimization on the test case. For the first ten times, the local cost simulation function will be turned off. Thus, all generated deployment strategies will be executed in the cloud. Then the rest ten times, the local cost simulation

function will be turned on to see if it will eliminate some pricey strategies before actually testing them. The results and analysis are described in Chapter 6.

5

Design and Implementation

This section first describes the design of the multi-level deployment optimization (**MLDO**) framework in general. Then, a detailed explanation of each component will be presented. Furthermore, the FaaS application fusion handler will be presented. At last, the FaaS application for benchmark will be introduced. The source code is available on Github.[1]

5.1 Overview of MLDO Framework

5.1.1 Fundamental Design Decision

During the literature review phase, we noticed that most existing solutions focus on resource allocation[11, 12, 29]. We agree that optimizing memory settings will effectively improve the performance and therefore reducing execution duration. However, when the optimization task advanced to application-level, simply optimizing resource allocation is not enough. According to the comprehensive report [13], based on the large dataset has been investigated, the authors stated cold start takes up an unignorable part of the total execution time for most applications (75%), and they argued that "*this makes avoiding and/or optimizing cold starts extremely important for the overall performance of a FaaS offering*," For example, considering there is an application with several functions, where each function has been deployed in a FaaS unit by itself. When invoking this application, each separated function will suffer from a cold start. These cold starts might take up a significant amount of execution time (latency). If we could combine some of the related functions (i.e., functions in the same business logic flow, or sharing same external libraries) into fewer deployment units, we can then reduce the amount of the cold starts, and eventually reduce the total execution time. Therefore, in our framework, we decide to have a dedicated latency (cold start) reduction module to cope with this problem.

There were a few potential solutions when we designed the framework during the first iteration: 1) a practical guideline of what should be done and avoided; 2) a framework that can test the most potential deployment strategies and select the best; 3) a framework that predicts the optimal deployment strategies with using a pre-trained model.

First, we ruled out potential solution 1, guideline, simply because such a list might

[1] https://github.com/dev-jp/mldo

be too abstracted and not able to fit all different applications. Nowadays, most FaaS users deploy the application in the finest granularity, i.e., each function in a separate deployment unit (container). Both the FaaS providers and industry practitioners have already provided suggestions on optimizing the deployment from resource allocation perspectives and reducing cold starts [30, 31, 31, 32].

Second, we decided to pick testing over predicting for this study. By predicting means the framework predicts the optimal deployment strategy by applying a pre-trained machine learning model. By testing means the framework tests the most potential deployment strategies and select the optimal one. In this study, we chose the testing method as the fundamental logic of our framework because of the following reasons. First, we believe that predicting is suitable for resource allocation tasks. However, how well the predicting method performs on the fusion tasks depends on the FaaS applications. It might be difficult for such a model to capture the application's internal structure (e.g., lib dependencies). Second, we found that using a pre-trained model may not be suitable for the frequently upgraded FaaS platform. Nowadays, the commercial FaaS platforms frequently upgrades the hardware and software to achieve better performance. Such infrastructure changes may not be captured by the pre-trained model. One may end up training the model again and again to keep the model up-to-date. Third, the study shows that 95% FaaS application contains at most 6 functions, of which 80% applications have at most 4 functions [14]. With such a manageable amount of functions, it is more economic to perform testing. Although, we selected the testing method, we still believed the statistical prediction method is feasible for resource allocation tasks. Thus, we designed our framework using a pipeline pattern that allows us to easily merge a statistical algorithm filter in the future. The framework will perform two levels of optimization, resource allocation task and latency reduction task. Each task can be performed separately.

Dynamic approach vs lock mocks. Dynamically deploying and testing the application in a black-box manner in the cloud was selected over the other two solution candidates, prediction based on local execution results, and statistical modeling based on empirical measurements. The main reason that stopped us from considering using local mocks is that a FaaS application could not run locally on a mock because some dependent external services are not available locally [33]. Also, all mocks are developed by third parties, e.g. Serverless[2], Thundra[3], etc. Such mocks may behave differently to the public platform. Furthermore, the rapid upgrade of the hardware and/or runtime of the FaaS platforms makes it challenging to reflect the objective performance measurement on a mock that simulates an older version of the FaaS platform. We believe mock is helpful for unit testing that focuses on business logic evaluation but is not suitable for an integration level evolution like our study. Lastly, the local mocks usually require additional skills, which will introduce a learning curve for inexperienced users.

[2]https://www.serverless.com/
[3]https://www.thundra.io/

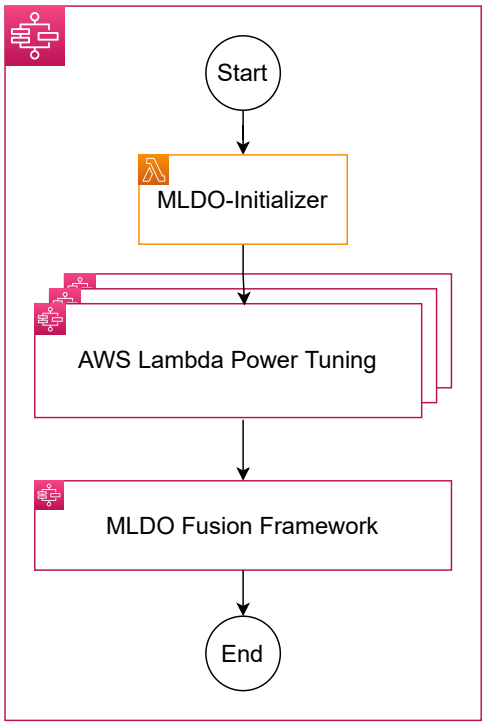


Figure 5.1: MLDO Framework Architecture

Automation. Automation is one of the goals we hope our framework could achieve. Deploying a FaaS application with multiple functions in different strategies requires extra human effort. We hope the framework we implemented can automate the tedious configuration task, i.e., the generating and testing multiple memory and application layout configurations. We design the framework to require users for minimal inputs, including information of the functions, invoke orders, payloads, and memory testing options.

5.1.2 High Level Architecture of MLDO Framework

As illustrated in Figure 5.1, the Multi-Level FaaS Application Optimization Deployment framework (**MLDO**) contains three components: MLDO-Initializer, Modified AWS Lambda Power Tuning, and MLDO Fusion Framework. These three components are chained into a "pipeline" model using AWS Step Functions[19]. Among the three components, AWS Lambda Power Tuning and MLDO Fusion Framework are wrapped as standalone state machines and can be run separately. Worth mentioning, although MLDO Framework is written in JavaScript, it is language agnostic.

First, the MLDO framework starts with MLDO-initializer, a simple Lambda function that takes JSON data with FaaS application structure and parameters for AWS Lambda Power Tuning Tool. It prompts users for initial input and passes the processed data to the next state. Users shall provide a JSON file with the structure of the FaaS application and "power values," a set of predefined memory values to test as input.

Then, the processed data out of MLDO-Initializer will be passed to the modified AWS Lambda Power Tuning, the memory optimization module that can run on multiple Lambda functions simultaneously. The modified AWS Lambda Power Tuning will first perform resource allocation tests on each Lambda function, record the execution data, calculate the optimal memory size, and wrap and return the optimal memory value with other execution data (execution duration). The AWS Lambda Power Tuning was developed by Alex Casalboni and hosted on GitHub as an open source software [10]. As described in related work, the tool can only perform memory allocation on a single Lambda function. We modified it by applying AWS Step Functions to make it spin up multiple instances and run on each function of an application simultaneously. More details will be presented in Section 5.2.

Last, the MLDO fusion framework, a module that can dynamically deploy functions based on given deployment strategies. The MLDO fusion module will first generate numerous possible deployment strategies by using output data (optimal memory value and execution data of each function) from AWS Lambda Power Tuning, together with the FaaS application structure. Then, the fusion module will follow the deployment strategies to deploy the application dynamically. After multiple tests on the application, the fusion module will analyze the execution logs and return the optimal deployment strategy in terms of speed, cost, and trade-off balanced. More details regarding this fusion framework will be presented in Section 5.3.

What is more, to enable "dynamic deployment" to a FaaS application, there is a fusion handler in each Lambda function. The fusion handler will receive a deployment strategy and perform the dynamic deployment. This fusion handler is discussed in Section 5.4.

5.2 Modified AWS Lambda Power Tuning

The AWS Lambda Power Tuning (**PT**) was selected as the solution to cope with memory optimization. Two factors have been taken into account, functionality and development cost. First and most important, AWS Lambda Power Tuning is recognized as an easy to deploy and widely adopted tool to analyze and optimize the memory settings of a single Lambda function. It might not be as "smart" or "efficient" as other strategies, e.g. Bayesian Optimization used by COSE [11], or pre-trained models used by Sizeless [12]. However, this most straightforward logic provides a fundamental solution for all kinds of Lambda functions. AWS has proved its functionality and listed it on AWS Serverless Application Repository for users to quickly deploy and run [29]. With our modification, it can meet the requirements of MLDO framework. Second, besides providing required functionality, AWS Lambda Power Tuning is an open-sourced, which enables us focusing more on developing other components.

Figure 5.2 is an output example of AWS Lambda Power Tuning. It visualized the cost and speed for each memory configuration of the four functions of test case 2

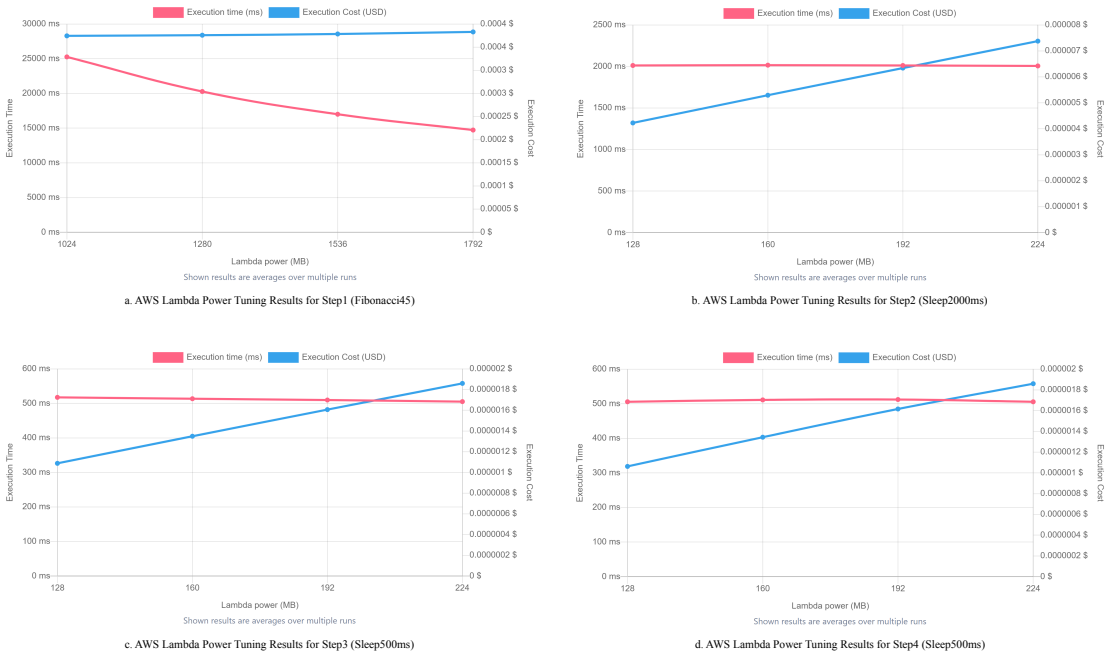


Figure 5.2: AWS Lambda Power Tuning's Results of the 4 functions of Test Case 2

of one execution during iteration 2. In the figures, the X-axis represents memory options, while Y-axis represents execution time. The blue and red curves illustrate the execution cost and the execution duration with different memory allocated respectively. For example, in the top left graph a, the red curve shows the execution time of function Step1 drops gradually from about 25s with 1024MB to less than 15s with 1792MB. Meanwhile, the blue curve shows the execution cost just goes up slightly as allocated memory increases. The optimal memory setting for function Step1 in this case is 1792MB. Graph b, c, and d represent the results of function Step2, Step3 and Step4, displaying different trends as Graph a. The execution costs of these three functions go up steadily with gradually increasing memory, but execution costs remain almost the same. Therefore, the optimal memory settings for these three functions are all 128MB. Such a outcome from AWS Lambda Power Tuning proves, when with static payload, it is capable of providing what we expected.

The MLDO framework aims to optimize applications with multiple functions rather than a single function. Hence, we use the "Map/Iteration" state features of AWS Step Functions to enable AWS Lambda Power Tuning to run tests on all functions of the given application in parallel. Adopting the "Map/Iteration" state instead of modifying the original source code of AWS Lambda Power Tuning frees us from upgrade concerns in the future. AWS Lambda Power Tuning is widely used and well maintained by the open source community. It has been regularly upgraded. For example, On December 1st 2020, AWS updated the pricing policy of Lambda memory [23], AWS Lambda Power Tuning was upgraded accordingly two days later.

Figure 5.3 illustrates the modified architecture of the AWS Lambda Power Tun-

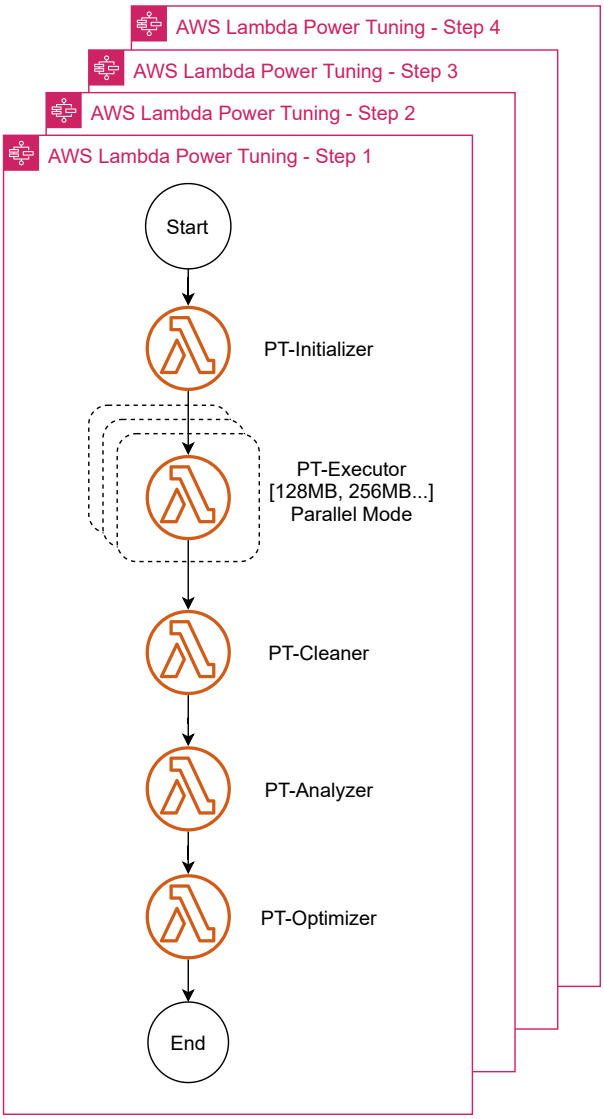


Figure 5.3: Architecture of the Modified AWS Lambda Power Tuning (PT) Module

ing. This module is designed to work as follows. First, it takes the structure of the application and "power value" settings of each function as the input from the previous module "MLDO-Initializer". Then, the "Map/Iteration" feature of AWS Step Functions splits the input and generates multiple instances of "AWS Lambda Power Tuning" so that each instance can run tests on one particular function. Last, it combines the invocation results of all functions and outputs optimization memory values for each function, which will be used in the subsequent module, "MLDO-Fusion".

5.3 MLDO Fusion Framework

The fusion module is the core of the MLDO framework. It is assembled with four

Lambda functions: Fusion-Initializer (**F-Initializer**), Fusion-Executor (**F-Executor**), Fusion-Cleaner (**F-Cleaner**) and Fusion-Analyzer (**F-Analyzer**). The fusion module will first generate multiple deployment strategy candidates with predefined constraints. Then, the application will be dynamically deployed and tested with synthetic payloads. Afterwards, the execution results will be analyzed, and the optimal deployment strategy will be picked out. Figure 5.3. presents the architecture of the MLDO Fusion framework.

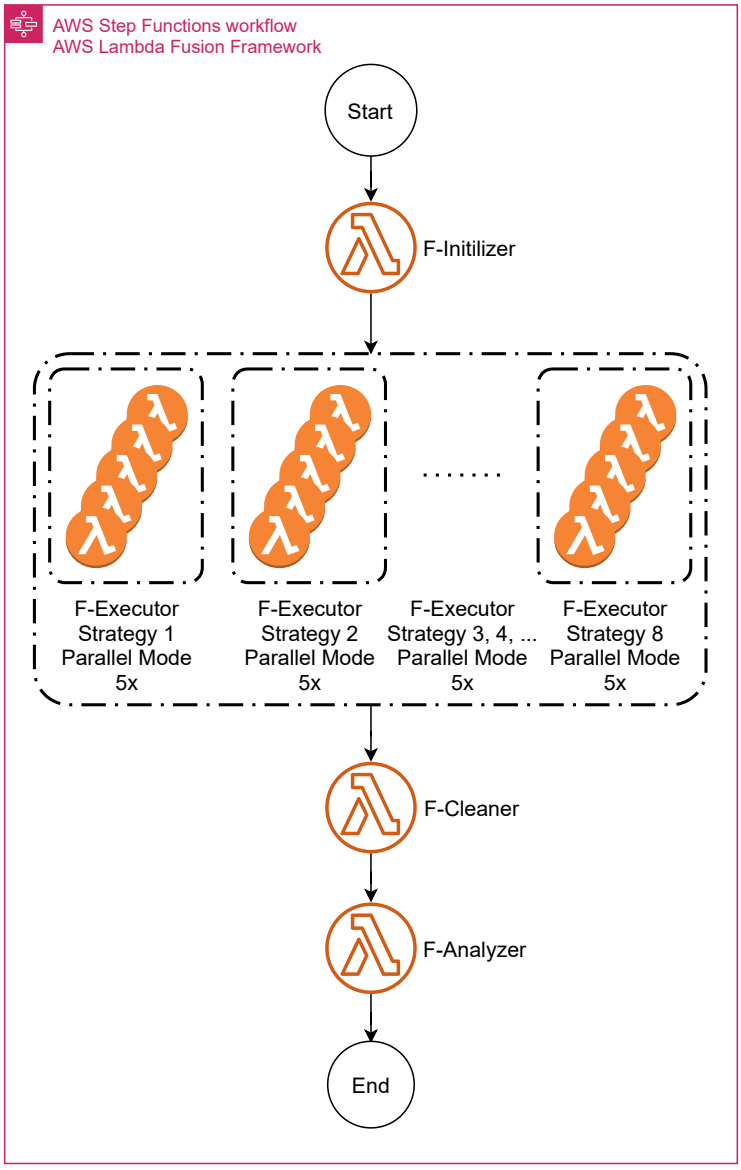


Figure 5.4: Architecture of the MLDO Fusion Module

5.3.1 F-Initializer

F-Initializer is the entry state of the fusion framework. It takes the application's structure with optimal memory values as input from the previous module, "AWS Lambda Power Tuning". First, the input data will be processed and filtered with pre-

defined constraints to generate multiple deployment strategy candidates. Table 5.1 lists all predefined constraints used in this framework. All constraints are designed to be formed into a pipeline model, which leaves the possibility of adding/removing constraints in the future. All generated strategy candidates will first run a local cost simulation test before actually testing them. By using the output from AWS Lambda Power Tuning, the module estimates the total costs of each strategy and removes the outstandingly expensive deployment strategies. Due to the time constraint, F-initializer currently is only able to run cost simulation. Afterwards, F-Initializer also assigns a unique traceID to each of the remaining strategy candidates. And this unique traceID will be passed along to subsequent functions. After the FaaS application is to be deployed as deployment strategy, the traceID will be logged when the application is invoked, and exact logs with this traceID will be filtered out from AWS CloudWatchLogs and analyzed later on. At last, the F-Initializer model will create separate Lambda aliases/versions for each strategy. The aliases/versions will be used specifically for each strategy.

5.3.2 F-Executor

Taking the deployment strategy candidates from F-Initializer, we split the candidates by adapting the “Map/Iteration” of AWS Step Functions. Each strategy candidate is passed into a dedicated instance of F-Executor. All instances of F-Executor will run in parallel. With the deployment strategy, F-Executor will first find the “root” function of the target FaaS application and invoke that "root" function. Each deployment strategy will be tested multiple times in parallel. The payload is handled by the fusion handler in the FaaS application, which will be described in Section 5.4

5.3.3 F-Cleaner

As its name suggests, the F-Cleaner module is designed to clean the aliases/versions created by the F-Initializer after testing or error.

5.3.4 F-Analyzer

F-Analyzer is designed to retrieve the execution logs of a FaaS application and analyze the logs to come up with the optimized deployment strategy in terms of “cheapest”, “fastest”, or “balanced”. The cheapest strategy is the one with the lowest execution cost disregarding its performance. The fastest strategy is the one with the shortest execution time disregarding its cost. Furthermore, the balanced strategy is a compromise strategy between "cheapest" and "fastest" based on the parameter "Weight", which is a parameter between 0.0 and 1.0 (by default 0.5), that express the trade-off between cost and time, 0.0 is equivalent to "fastest" strategy, 1.0 is equivalent to "cheapest" strategy.

As described in Section 5.3.1., each strategy candidate has a unique traceID. The traceID will be logged when the function is invoked. All the execution logs are stored with AWS CloudWatch by default [34]. F-Analyzer can retrieve the execution logs and group them by traceID, then run analysis.

Table 5.1: Constraints applied when generating deployment candidates.

Constraint	Descriptions	Examples
Duration 15mins	Lambda has a hard limit of execution duration of 15 mins [18]. Hence the execution duration of each deployment unit can not exceed 15 mins. Otherwise, Lambda will throw an "execution timeout" exception.	Assuming a deployment unit has two functions, both functions have a proximate 10 mins duration. The total estimated duration is 20 mins which exceed the Lambda timeout limit.
Jump Back	With a distributed serverless system such as AWS Lambda, it is difficult to reuse the instance once it finishes its work due to the lack of traceability of the instance. One possible way is to keep the instance alive on purpose, then synchronously invoke the following unit and wait for the response. However, this method will cause the "double billing" problem, which is anti-pattern [30]. Hence, we should remove all candidates that have "jump back" occurrence.	Appendix A: #P09, #P10, #P11
Double Entries	Same reason as hehind "No Jump Back" constraint, it is impossible to guarantee the same instance will be invoked when more than one invocation requests targets to this unit. On AWS Lambda, when there are multiple requests sent to one unit, Lambda will generate multiple instances to handle each request. Hence, we should remove all candidates that have "double entries" occurrence.	Appendix A: #P10, #P12, #P13, #P14, #P15
Memory	Memory constraint is rather a strategy than a constraint. When fusing a few functions into one deployment unit, the memory allocation depends on if there is a shared resource among the functions, e.g. same external library, same variables, etc. However, it is difficult to determine without a code-level analysis, and it is beyond the scope of this study. In this study, we allocated the unit's memory with 1.0x of the biggest memory requirement of a function.	A deployment unit has two functions, "step1" with a RAM requirement of 200MB and "step2" with a RAM requirement of 300MB. The deployment unit will be allocated with 300MB * 1.0x = 300MB.

5.4 MLDO Fusion Handler

In order to run a dynamic deployment strategy, the FaaS application must be able to process the deployment strategy data and invoke the target functions either locally or remotely. Thus, We designed a fusion handler to cope with the deployment task. The fusion handler is wrapped in a separate file, thus it has no impact on the structure of the source code of the FaaS application. Figure 5.5 illustrates how the fusion handler works. Unlike the MLDO framework, the fusion handler is not language agnostic. The fusion handler works inside the FaaS application, hence the fusion handler has to be written in the same language as the FaaS application. In this study, we implemented the fusion handler in JavaScript.

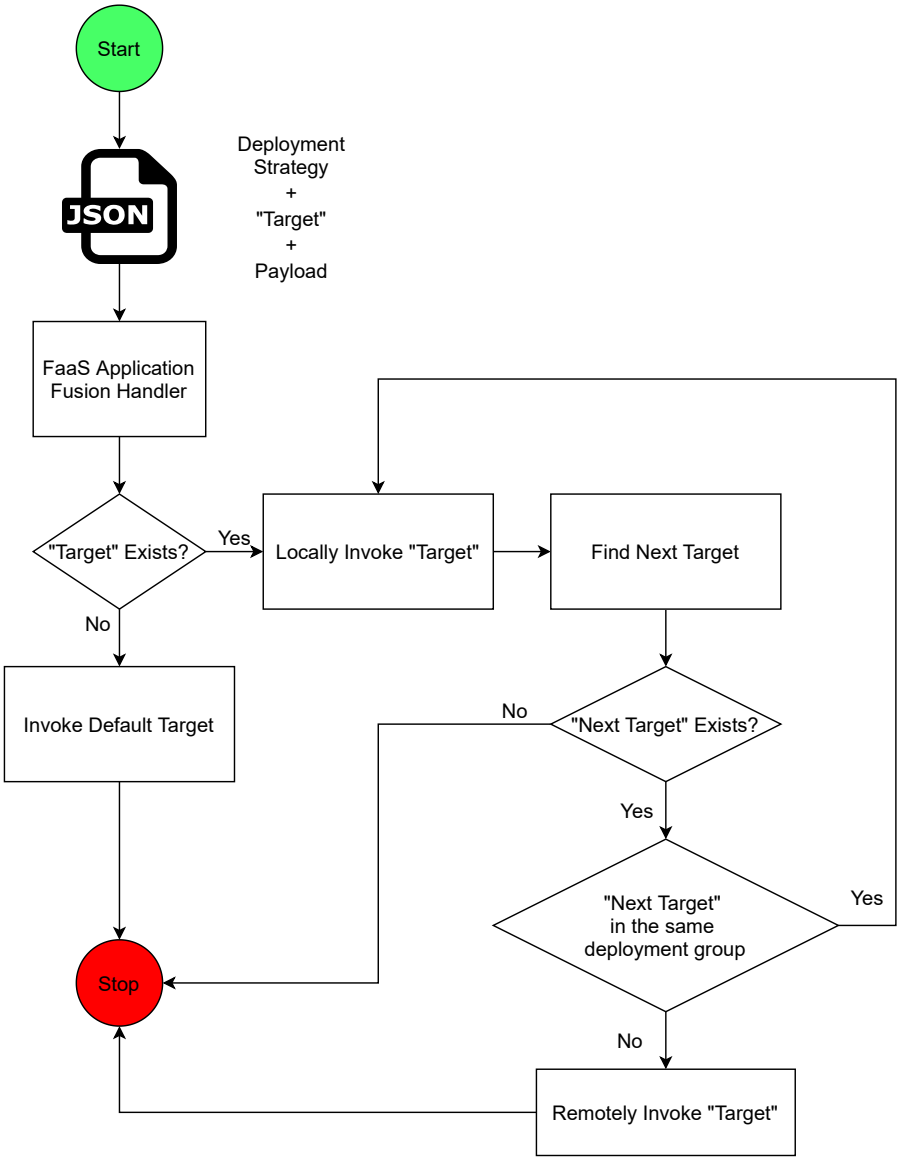


Figure 5.5: MLDO Fusion Handler Workflow.

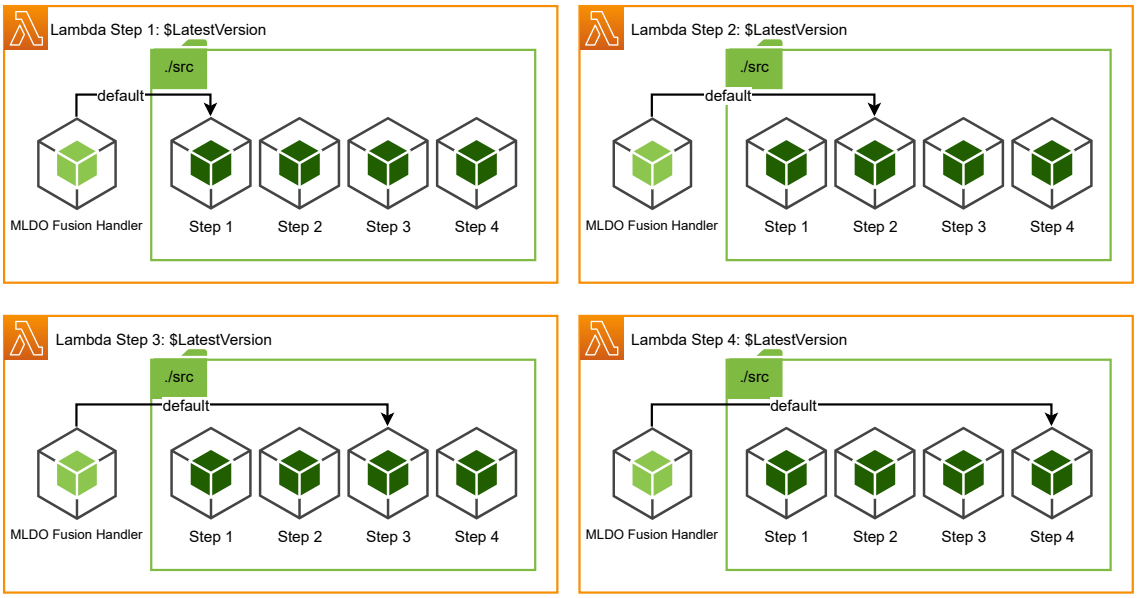


Figure 5.6: Example FaaS Application Initial Deployment with Finest Granularity.

5.5 FaaS Application for Evaluation

A test case that represents real-world applications is essential to this study. Such a test case will be used through design and implementation, and in the final evaluation phase. With the fast development and adoption of FaaS, thousands of applications have been deployed and running on the FaaS platforms nowadays. Each application will contain multiple functions. Unfortunately, there is no real-world ready test case to use out there. No benchmarks claim their test cases are well built and can represent realistic applications. Exploring the characteristics of real-world FaaS applications is a broad topic and out of the scope of this study. However, a few studies were conducted on large amounts of real-world FaaS applications, summarized the characteristics of FaaS applications [7, 14], and helped us gain a deep understanding of what real-world FaaS functions look like. For example, Shahrad et al., after investigating a significant dataset and stated that most FaaS applications contain between 3 to 6 functions, of which 80% of applications have at most 4 functions [14]. Thus we abstracted a real-world image processing application that contains 4 functions and both sequential and parallel workflows to structure the test case (Figure 1.1).

Figure 5.6 illustrates the default deployment mode, each function of the FaaS application test case is deployed as separate Lambda function. When a deployment strategy is received, the fusion handler will dynamically fuse the functions. An example as shown in Figure 5.7, given a deployment strategy which divides the functions into two groups: one group with “step1”, "step2" and “step4”, and the other group only with “step3”. In this case, only two Lambda unit with specific versions (“Alias124-3”) are required, namely Lambda Step 1 and Lambda Step 4. Inside the Lambda unit "Step 1", the fusion handler first invoke Step 1 locally, then

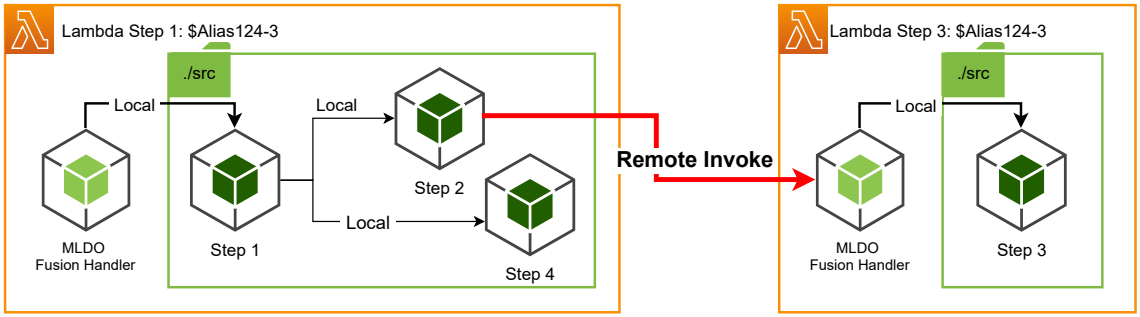


Figure 5.7: An Deployment Example of A FaaS Application.

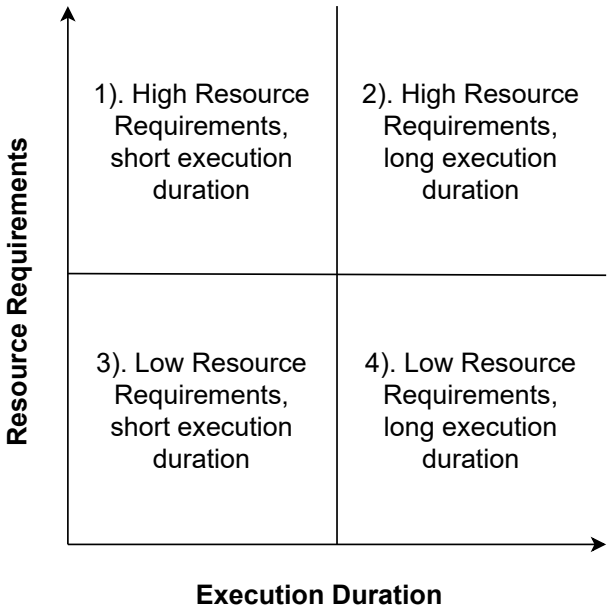


Figure 5.8: 4 Types of FaaS Function In Terms of Required Resource and Execution Duration.

locally invokes Step 2 and Step 4 simultaneously. Once Step 2 finishes execution, the fusion handler will invoke Step 3 remotely.

We categorize the FaaS functions into 4 types from the perspectives of pricing model of FaaS platforms (i.e., resource required and execution duration). They are presented in Figure 5.8, namely 1) high resource requirements and short execution duration, 2) high resource requirements and long execution duration, 3) low resource requirements and short execution duration, and 4) high resource requirements and long execution duration.

We use Fibonacci number calculation to control resource allocation (List 1). Figure 5.9 shows the performance when calculating the 45th Fibonacci number with different memory values. As the figure illustrates, the calculation requires a large amount of memory to perform well. With 128MB, it takes more than 200 seconds, and it reduces to about 15 seconds when memory increases to 1792MB. It is worth men-

```
const getFibonacci = N => {
    if (N < 2) {
                return 1;
        } else {
                return getFibonacci (N-2) + getFibonacci (N-1);
        }
}
```

Listing 1: Code Snap of Fibonacci Number Calculation.

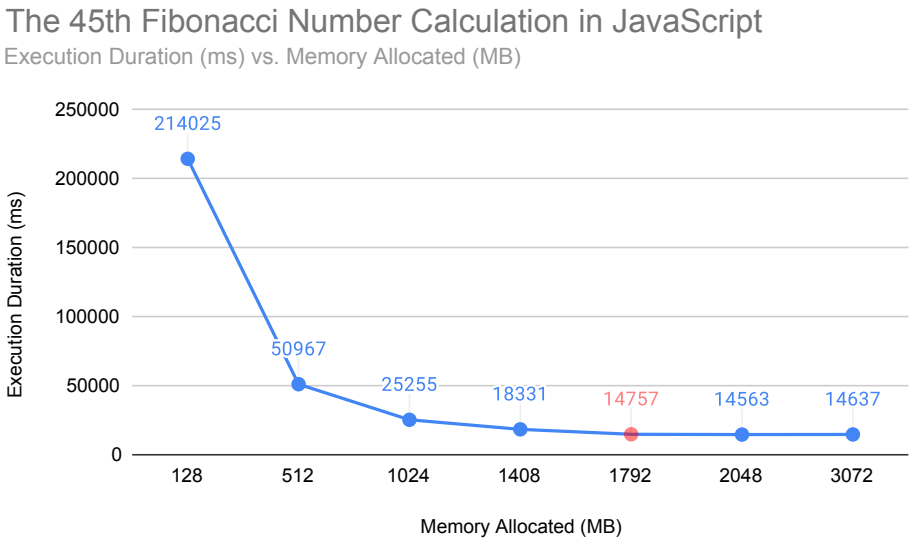


Figure 5.9: AWS Lambda Performance of the 45th Fibonacci Calculation in JavaScript

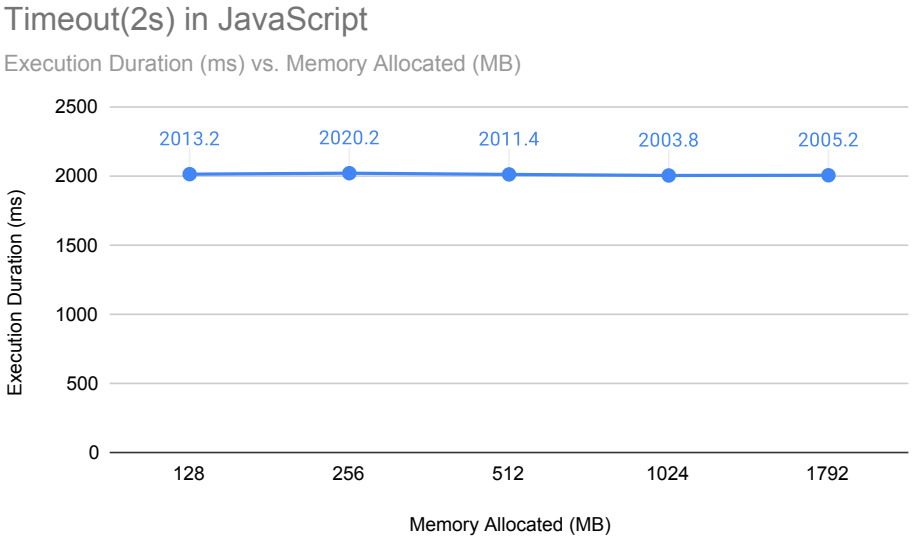


Figure 5.10: AWS Lambda Performance of Timeout 2 seconds in JavaScript

tioning that the performance does not improve when we keep increasing the memory from 1792MB to 3GB. That is because Node.js is a single-threaded programming language, and it can only run on a single CPU core which on AWS Lambda equals 1792MB. To control the execution duration, we use the timeout method. Figure 5.10 presents the performance when a function timeouts 2 seconds with different memory values. As expected, changing memory value has almost no impact on the performance. By mixing the Fibonacci calculation method and timeout method, we can simulate the four types of functions shown in Figure 5.8.

6
Evaluation and Discussion

To answer RQ3, we designed and carried out a standalone controlled experiment to evaluate the artifact. This section outlines the experiment setup, presents and discusses the evaluation results, draws its implications, and summarizes the findings for RQ3. In the end, we also discuss the validity threats to this study.

RQ3: How effectively does the new multi-level FaaS application deployment optimization framework optimize the deployment?

6.1 Experiment Setup

In Section 1, Figure 1.1 presents the structure of the test case through this thesis. By mixing different types of functions, we simulate a few different scenarios. Table 6.1 listed the test cases that have been used at different phases of this thesis study. The top three test cases were used for development in the three iterations, while the last test case was carried out specifically for the final evaluation. The following subsection will present the results of last test case.

The testing process goes as following steps. First, the MLDO framework deploys and executes the test case application according to each deployment strategy. Then, it measures the time elapsed for running the test case and calculates the total cost, as well as the trade-off balanced value. By comparing the total cost, execution elapsed time, and trade-off balanced value of each deployment strategy, the framework determines which strategies are optimal over others.

We perform 20 times of deployment optimization on the test case. For the first 10 times, the local cost simulation function will be turned off. Thus, all generated deployment strategies will be executed in the cloud. Then for the rest 10 times, the local cost simulation function will be turned on to see if it will eliminate some pricey strategies before actually testing them.

Afterwards, the results of these 20 tests will be collected and analyzed. We will examine if the framework successfully picks out the optimal strategies in terms of cost, speed, and trade-off balanced as expected. We will also check if the data is consistent between local cost simulation on and off.

The MLDO framework and test case application are deployed in the "eu-north-1"

region (Stockholm). All 20 times were carried out continuously in about 8 hours. Each test took about 5 mins, and we deliberately waited for 10 to 15 mins to avoid AWS SDK throttling.

Table 6.1: Test Case Design

Phase	Type	Functions Description	Expectations
Iteration 1	Parallel workflow, Mixed types of functions, Static payload	**Step1** sleep(10s), 128MB **Step2** sleep(5s), 128MB **Step3** sleep(2s), 128MB **Step4** sleep(2s), 128MB	If fusing all functions into one Lambda unit without reaching the timeout limit (15 mins), it's believed that fusing all functions together is the most optimal deployment strategy.
Iteration 2	Parallel workflow, Mixed types of functions, Static payload	**Step1** sleep(5s) **Step2** Fibonacci(45) **Step3** sleep(2s), **Step4** sleep(2s)	Step2 demands a large amount of computing power. Other functions should not be fused with Step2.
Iteration 3	Parallel workflow, Mixed types of functions, Controlled, Static payload	**Step1** sleep(5s), 1792MB **Step2** sleep(2s) **Step3** sleep(0.5s) **Step4** sleep(0.5s)	- Eliminate the interference of the dynamical resource limits of the FaaS platform. - The upgraded framework should rule out a few strategies during local cost simulation.
Evaluation	Parallel workflow, Mixed types of functions, Static payload	**Step1** Fibonacci(45) **Step2** sleep(5s) **Step3** sleep(0.5s) **Step4** sleep(0.5s)	-The upgraded framework should rule out a few strategies during local cost simulation. - The fastest strategy should be fusing all or most functions, which aims to reduce cold starts. - The cheapest strategy should leave Step1 alone due to its large demand for memory than others. - The trade-off balanced strategy depends on the weight parameter.

6.2 Results

Figures 6.1, 6.2, 6.3, and 6.4 present the final evaluation results regarding execution elapsed time, total cost, and trade-off balanced value. The corresponding data is available on Github. As planned, there were 20 tests carried out for the evaluation. The local cost simulation was off for the first 10 tests. Thus we see the MLDO framework generated 8 possible deployment strategies in Figures 6.1, 6.2, and 6.4. For the last 10 tests, the local cost simulation was on to help eliminate some outstanding strategies (in this case, the pricey strategies). Thus we only see 4 possible deployment strategies that were generated in Figure 6.3.

In Figure 6.1, the X-axis (horizontal) categorizes the 8 different deployment strategy candidates, while the Y-axis (vertical) displays the execution elapsed time of the test case in milliseconds. The violin plots represent the elapsed time of the test case deployed with different strategies. According to the chart, strategy [S1, S2, S3] - [S4] is the fastest one, the expected strategy [S1, S2, S3, S4] comes the second fastest with about 300ms difference. The slowest strategies are [S1, S4] - [S2] - [S3] and [S1] - [S2] - [S3] - [S4], which are about 2s slower than the fastest one. The result shows that tests with more deployment units tend to consume more time, which makes sense that the initial cold start is reduced by fusing functions. According to the average data, the fastest strategy runs 10.5% faster than the slowest strategy. However, we see an exception that the strategy "[S1, S2, S3] - [S4]" costs less time than the strategy "[S1, S2, S3, S4] to the left. This is due to the dynamic resource variation of the AWS Lambda platform, and it will be discussed in the following subsection.

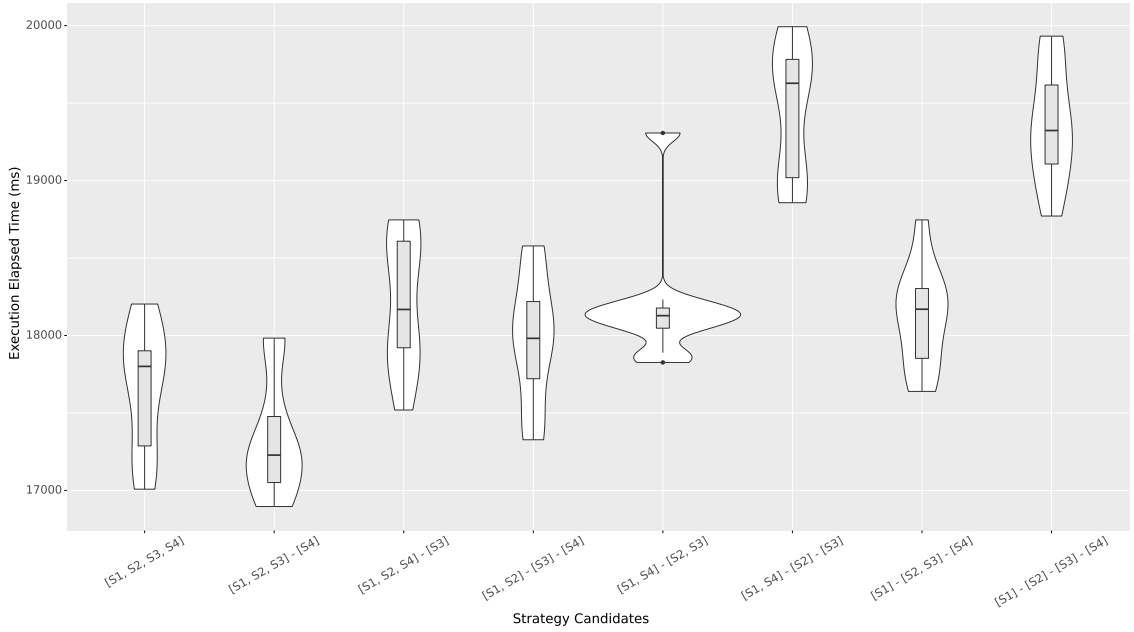


Figure 6.1: Results of Execution Elapsed Time

In Figure 6.2, the X-axis (horizontal) categorizes the 8 different deployment strategy

candidates, while the Y-axis (vertical) displays the execution cost in USD. The violin plots represent the total cost of the test case deployed with different strategies. In the test case, we designed S1 as a long execution and high resource demanded function, we expected to see strategies with no functions fused with S1 outrun others, and the more functions were fused with S1, the more expensive it will be. Results shows that without local cost simulation on, strategy [S1] - [S2, S3] - [S4] is the cheapest one, and strategy [S1] - [S2] - [S3] - [S4] is second cheapest with a tiny difference. Figure 6.3 shows that the local cost simulation successfully picked out the most expensive strategies (left four strategies in Figure 6.2). By cross-checking with the execution elapsed time chart (Figure 6.1), we see a trend that the less time it consumes, the more it likely costs. This is due to fusing low resource demand functions (i.e., S2, S3, S4) with high resource demand functions (i.e., S1). According to the average result, the cheapest strategy runs 13.3% cheaper than the most expensive strategy.

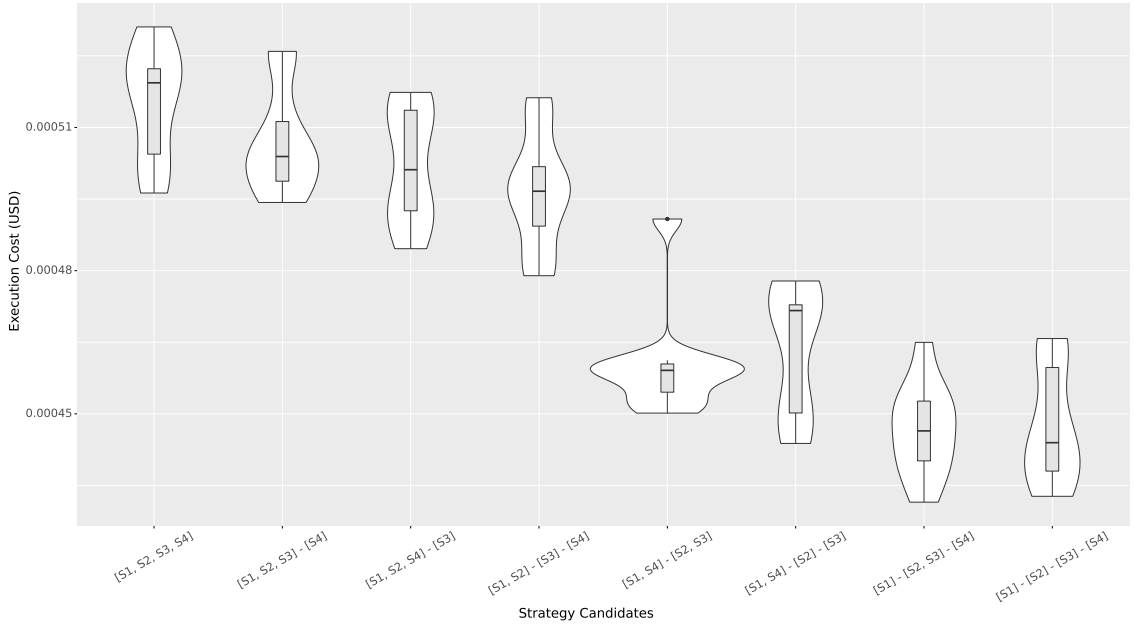


Figure 6.2: Results of Total Execution Cost without Cost Simulation Function on

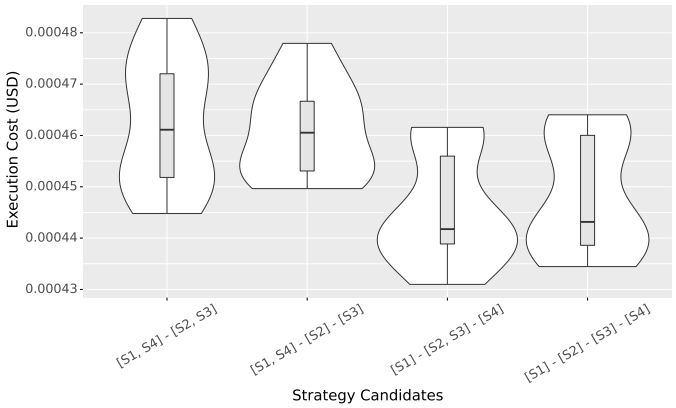


Figure 6.3: Results of Total Execution Cost with Cost Simulation Function on

Last, in Figure 6.4, the violin plots represent the trade-off balanced values calculated on the given weight (0.5 in this case). It helps to find out a compromise strategy between "cheapest" and "fastest" based on "Weight." Weight is a parameter between 0.0 and 1.0 (by default 0.5) that expresses the trade-off between cost and time. 0 is equivalent to "fastest" strategy, 1 is equivalent to "cheapest" strategy. The balanced values are calculated like below:

$$Balanced = Weight\frac{Cost}{MaxCost} + (1 - Weight)\frac{Elapsed}{MaxElapsed}$$

The strategy with the smallest balanced value is the strategy that fits the desired trade-off requirements. The curve clearly shows that the cheapest strategy [S1] - [S2, S3] - [S4] is also the most balanced strategy in this case (Weight = 0.5).

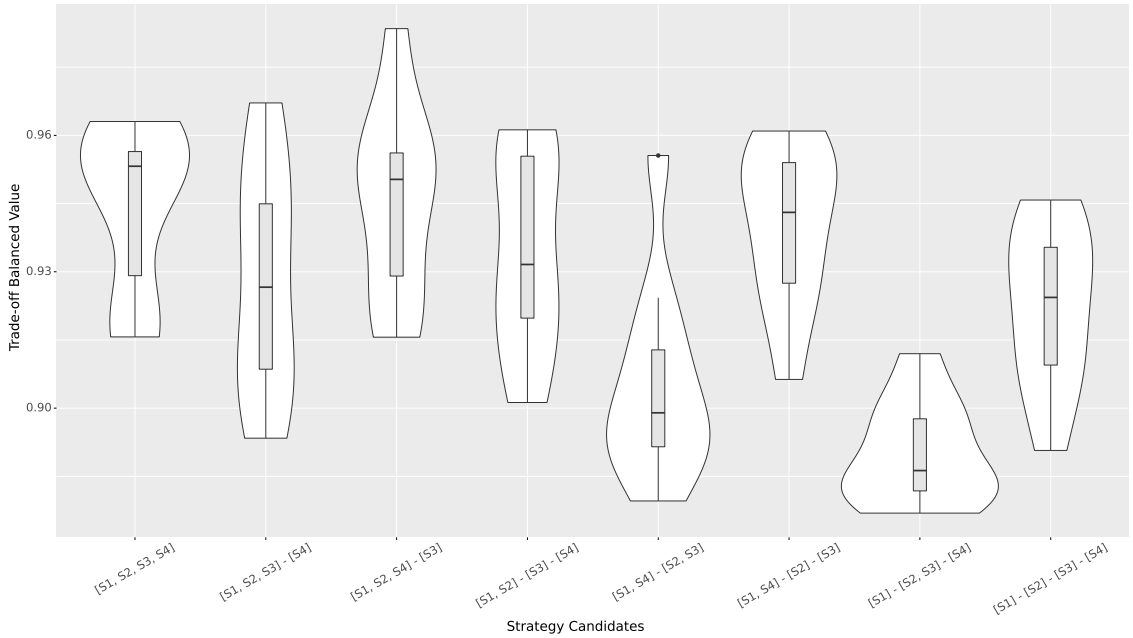


Figure 6.4: Results of Trade-off Balanced Value (Weight=0.5)

6.3 Discussion

In general, the evaluation results shown in Figures 6.1, 6.2, 6.3, and 6.4 are promising and meet our expectations. The framework successfully picked out the optimal deployment strategies in terms of “cheapest” and “trade-off balanced” and returned the expected fastest strategy as the second with little difference to the "fastest" one. We also noticed that the local cost simulation function successfully removed potentially expensive deployment strategies without actually testing them in the cloud, which can effectively reduce workload and improve general performance. Furthermore, the trade-off balanced strategy can help users to deal with more complicated scenarios.

6.3.1 Dynamic resource variation

When analyzing the result, we noticed a few exceptions. For example, the expected fastest deployment strategy ran slightly slower than the actual fastest one. We studied the execution logs of these exceptions and found that the execution time consumed by the function (in this case, S1) calculating the 45th Fibonacci number is not consistent. We believe it is due to the dynamic resource variation of the AWS Lambda platform. With the same memory allocation, the execution duration of this function in some cases is considerably longer than the others. To understand more about this problem, we further studied 10 tests. Table 6.2 shows 4 example execution data of S1, which is the function that calculates the 45th Fibonacci number with 1792MB allocated. For this particular FaaS application used for evaluation, there are only 2 potential deployment strategies that have S1 deployed by itself, [S1] -[S2] - [S3] - [S4] and [S1] -[S2, S3] - [S4]. The table shows that with the same payloads, the execution duration varies from 14526 ms to 15626 ms with a significant difference of 1100 ms. Meanwhile, the cold start varies from 379.37 ms to 504.72 ms with a difference of 125.35 ms. In the evaluation test case, S3 and S4 were designed as sleeping functions with 500ms timeout. In an unusual case, this 1100 ms gap could end up with S1 deployed alone running longer than S1 and S4 fused together, i.e., the former strategy ends up more expensive than the latter one. Such dynamic resource variation is out of our control and will dramatically affect the optimization results of a test case like the one used in the evaluation. To cope with this issue, a recommendation is to run the optimization process multiple times and let the framework use the average result to analyze.

Table 6.2: Execution Data Examples of Function Step1

Strategy	Execution Duration (ms)	Cold Start (ms)
[S1] - [S2] - [S3] - [S4]	14600	379.37
[S1] - [S2] - [S3] - [S4]	14526	412.51
[S1] - [S2, S3] - [S4]	15626	445.44
[S1] - [S2, S3] - [S4]	15300	504.72

6.3.2 Scalability

In theory, there is no hard limit on how many the MLDO framework can test. The soft constraints come from the AWS platform. The MLDO framework has several methods that invoke AWS SDK, e.g., creating a new version of a function, allocating different memory options to a function, retrieving execution logs from AWS CloudWatch, etc. Occasionally, we encountered throttling problems. Even though we followed the suggestions that AWS gave [35], and designed the framework carefully, we sometimes hit invisible limits. Right now, we deliberately extend the waiting time between each stage of the optimization process to avoid triggering the throttling exception. Currently, the MLDO framework can test up to 50 functions, where this limit comes from a method retrieving execution logs from the AWS CloudWatch. If an application contains more than 50 functions, we have to split it up to below 50 and reduce invocation frequency to avoid the throttling problem. Besides

passively respecting the limit from the AWS platform, we also actively implemented the local cost simulation function during iteration 3 to reduce the overall workload and improve performance.

6.3.3 Generality

Worth mentioning, when designing the resource allocation module, instead of starting from scratch, we reused an open-source framework, AWS Lambda Power Tuning, over other solutions, e.g., COSE using Bayesian Optimization model to predict. We chose AWS Lambda Power Tuning because of its sub-exhaustive searching method is more suitable to our study comparing to training a model. However, we see the possibility of applying statistical prediction method, e.g., Bayesian Optimization in the future. Hence, to increase the generality, we designed the MLDO framework into a pipeline model, where both resource allocation module and fusion module work as a standalone component. This leaves great possibility and easiness to replace AWS Lambda Power Tuning with other solutions in the future.

6.4 Threats to Validity

In this section, we analyzed four types of validity threats categorized by Runeson and Höst [36], namely internal validity, external validity, construct validity and reliability.

6.4.1 Internal validity

We deployed and ran the artifact on a public FaaS platform, where many cloud environmental factors, such as evolving infrastructure or dynamic resource limits, may affect the performance of a FaaS application. These confounding factors are out of control at most times. For example, in one of our test cases, a function calculating the 45th Fibonacci number with 1792MB memory allocated usually takes 14.5s to 15s based on the observations of nearly a hundred executions. Occasionally, the same function may consume more than 15.5s or less than 14s, and in such cases, the result is different from the majority. As an experimenter, we have to identify such abnormal results and determine if they are caused by the FaaS platform by digging into the detailed execution logs. Furthermore, to mitigate the interference from the FaaS platform, we deliberately designed the test case without dependencies to any external services, which might cause uncontrollable results.

6.4.2 External validity

There are two threats to external validity. First, the artifact was designed and implemented for the AWS Lambda platform specifically, and it can not be re-used directly on other FaaS platforms. However, we studied the most popular FaaS platforms and found these platforms offer similar pricing policies to AWS Lambda. Hence, we expect the fundamental mechanism of the framework should apply to other FaaS platforms as well. Such extended work on other FaaS platforms is out

of the scope of this thesis. However, we released the source code to the open source community, and researchers can grab and modify it according to the target FaaS platform in the future.

Second, the test cases we use for development and evaluation are running on static payload. The result of optimal deployment strategy is applicable for the specific payload or relatively similar payload. It may come out with different results for the same test case but with different payloads. For example, in our test case, a function calculating the 45th and 5th Fibonacci numbers within a similar execution period requires dramatically different amounts of resources. The consumption of different memories will result in different deployment strategies. The strategy for the application with a function calculating the 45th Fibonacci number would be avoiding fusing other low resource required functions with this specific function; while the strategy for the same application with a function calculating the 5th Fibonacci number might be suggesting to fuse other functions with it to reduce cold start. To mitigate this threat is beyond the scope of this study. However, a theoretical solution could be dynamically deploying the application based on the payload, which requires a dedicated function to determine how heavy is the payload at run time.

6.4.3 Construct validity

There is one challenge to the construct validity: the cost calculation model of our artifact only sums up the execution cost of Lambda functions but not external services (e.g. database usage, network transmission, I/O operations) if there are any. However, we believe this threat does not affect the final results. Our artifact works by comparing the costs of different deployment strategies within a short period. During this short period, we assume the FaaS platform behaves stably; the differences in external service costs in different deployments should be negligibly small. Hence, we can ignore the costs of external services when comparing the total cost. In general, the outcome of our artifact should be interpreted relatively (i.e. costs comparison horizontally to other strategies) instead of absolutely (i.e. used the absolute cost value outside the dedicated test case).

6.4.4 Reliability

Running the test cases that have been applied in this study might not result in the same or similar as it presented here in the future. As mentioned previously, to increase their competency on the market, the FaaS providers never stop evolving their technology, e.g. upgrading infrastructures, introducing new technologies, etc., and frequently adjusting their pricing policy. Amazon recently made massive changes on AWS Lambda, including increasing the memory limitation from 3GB to 10GB with the incremental reduction from 64MB to 1MB and reducing the chargeable duration interval from 100ms to 1ms. Such changes could draw a dramatic impact on the result of reproduction. However, creating a new fusion framework in the study introduces a handy tool and, more importantly, exploring a feasible strategy.

As long as there is no cross-generation upgrade, the framework is believed to result in a consistent trend in optimizing the deployment.

7
Conclusion and Outlook

7.1 Conclusion

Through this thesis work, we designed and implemented a novel multi-level FaaS application deployment optimization artifact. The artifact manages to find optimal deployment strategies for FaaS applications from resource allocation and latency reduction perspectives with one shot. Compared to testing deployment strategies manually, the highly automated process helps release the FaaS developers from the tedious configuration tasks of FaaS deployment.

By following the design research methodology, we answered the three research questions raised in the beginning.

RQ1 is a question related to the design of the memory allocation module of the artifact. Among all solution candidates, we eventually choose AWS Lambda Power Tuning as the solution for the artifact. As a widely adopted and proven framework by the community, we found AWS Lambda Power Tuning is capable of providing the results we expected. Also, as an open source framework, it requires minimal effort to modify and merge it into our artifact. We described this module in detail in Section 5.2

RQ2 is also a question regarding the design of the artifact. After selecting the design solution for the memory allocation module, we decided to use AWS Step Function to combine the memory allocation module and the latency reducing module. As an internal tool provided by AWS, it provides great simplicity and compatibility when working with AWS Lambda. With its help, we implemented the artifact into pipeline mode, which leaves the possibility to add or modify the filters in the future easily. We elaborate on the architecture of the artifact in Section 5.1 and Section 5.3

RQ3 is related to the evaluation of the artifact. The whole study was carried out by three iterations, and we applied a TDD like development process to guide us through each iteration. To evaluate the artifact, a standalone evaluation has been carried out by the end and the results are presented and analyzed in Section 6. The final results confirmed that the novel artifact meets our design expectations.

Overall, the concrete contributions of both the prototype of the new framework and the adapted optimizing strategy have high relevance for both practitioners and

researchers. Furthermore, the source code and test results were released to the open-source community, hopefully getting attention for further study.

7.2 Outlook

First of all, we have noticed that running a local simulation can help reduce the payload and improve the overall performance from the final evaluation. However, due to the time limit, we only implement the local cost simulation. To extend this study, it is worthy to implement the local elapsed and trade-off balance simulations.

Secondly, as mentioned multiple times previously in Section 3.1, Section 4.2.1 and Section 6.4, we believe that applying specific statistical methods, e.g. Bayesian Optimization, to the memory allocation module will bring more optimal results than the current method. Right now, we applied "exhaustive" searching for the optimal memory configurations to our artifact. However, this method runs on a subset of possible configurations and returns sub-optimal results based on the inputs. In Dec. 2020, AWS updated the Lambda memory configuration options, which increased the max memory limit from 3GB to 10GB and reduced the interval from 64MB to 1MB. Such a change makes it impossible to test all configurations. Predicting based on statistical methods seems more feasible in the current situation.

Thirdly, another direction would be increasing the external validity by implementing and evaluating other FaaS platforms, e.g. Azure Functions from Microsoft, Google Cloud Function, etc. Azure Function has a similar billing policy to AWS Lambda, i.e. cost is calculated by execution time based on memory. We are confident that the artifact should produce similar results as on AWS Lambda. However, Google Function is a bit more complex, as its charges are calculated by execution time based on memory and CPU power. It is interesting to see how artifact works on other platforms.

Last but not least, as described in Section 6.4.2, the test cases used in this study only ran on the static payload. An extent of work can be implementing functions that can also optimize applications with dynamic payload.

Bibliography

[1] Cloud computing services - amazon web services (aws). https://aws.amazon.com. Accessed: 14-Jan-2021.

[2] Cloud computing services | microsoft azure. https://azure.microsoft.com/en-us/. Accessed: 14-Jan-2021.

[3] Cloud computing services. https://cloud.google.com/. Accessed: 14-Jan-2021.

[4] S Newman and Building Microservices. O'reilly media inc. *Building Microservices*, 2015.

[5] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[6] James Lewis. Microservices—java, the unix way. In *Proceedings of the 33rd Degree Conference for Java Masters*, 2012.

[7] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, 2020.

[8] Aws lambda - serverless compute - amazon web services. https://aws.amazon.com/lambda/. Accessed: 14-Jan-2021.

[9] The state of serverless. https://www.datadoghq.com/state-of-serverless/. Accessed: 14-Jan-2021.

[10] Alex Casalboni. Deep dive: Finding the optimal resources allocation for your lambda functions. https://bit.ly/37Od05c. Accessed: 14-Jan-2021.

[11] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. Cose: Configuring serverless functions using statistical learning. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 129–138. IEEE, 2020.

[12] Simon Eismann, Long Bui, Johannes Grohmann, Cristina L Abad, Nikolas Herbst, and Samuel Kounev. Sizeless: Predicting the optimal size of serverless functions. *arXiv preprint arXiv:2010.15162*, 2020.

[13] Janos Czentye, Istvan Pelle, Andras Kern, Balazs Peter Gero, Laszlo Toka, and Balazs Sonkoly. Optimizing latency sensitive applications for amazon's public cloud platform. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2019.

[14] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing

the serverless workload at a large cloud provider. In *2020 {USENIX} Annual Technical Conference*, pages 205–218, 2020.

[15] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[16] Joel Scheuner and Philipp Leitner. Transpiling applications into optimized serverless orchestrations. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 72–73. IEEE, 2019.

[17] Junze Lai. Feedback-driven function fusion in faas deployments.

[18] Aws lambda enables functions that can run up to 15 minutes. `https://amzn.to/3yOsq5g`. Accessed: 12-May-2021.

[19] What is aws step functions? - aws step functions. `https://docs.aws.amazon.com/step-functions/`. Accessed: 24-Apr-2021.

[20] Gian-Carlo Rota. The number of partitions of a set. *The American Mathematical Monthly*, 71(5):498–504, 1964.

[21] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. Video processing with serverless computing: A measurement study. In *Proceedings of the 29th ACM workshop on network and operating systems support for digital audio and video*, pages 61–66, 2019.

[22] Github. `https://github.com/`. Accessed: 14-Jan-2021.

[23] Aws lambda changes duration billing granularity from 100ms down to 1ms. `https://amzn.to/3AEdjfk`. Accessed: 12-May-2021.

[24] Joel Scheuner and Philipp Leitner. Function-as-a-service performance evaluation: A multivocal literature review. *Journal of Systems and Software*, 170:110708, 2020.

[25] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[26] Nikhila Somu, Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Panopticon: A comprehensive benchmarking tool for serverless applications. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 144–151. IEEE, 2020.

[27] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. Faasdom: A benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, pages 73–84, 2020.

[28] Roel Wieringa. Design science as nested problem solving. In *Proceedings of the 4th international conference on design science research in information systems and technology*, pages 1–12, 2009.

[29] aws-lambda-power-tuning - aws serverless application repository. `https://amzn.to/3xNpzYV`. Accessed: 26-Apr-2021.

[30] Operating lambda: Performance optimization. `https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/`. Accessed: 26-Sept-2021.

[31] Aws lambda performance optimization. `https://lumigo.io/aws-lambda-performance-optimization/`. Accessed: 26-Sept-2021.

[32] Serverless computing: How to optimize aws lambda functions. https://www.cloudhealthtech.com/blog/aws-lambda-functions. Accessed: 26-Sept-2021.

[33] Valentina Lenarduzzi and Annibale Panichella. Serverless testing: Tool vendors' and experts' points of view. *IEEE Software*, 38(1):54–60, 2020.

[34] Amazon cloudwatch - application and infrastructure monitoring. https://aws.amazon.com/cloudwatch/. Accessed: 12-May-2021.

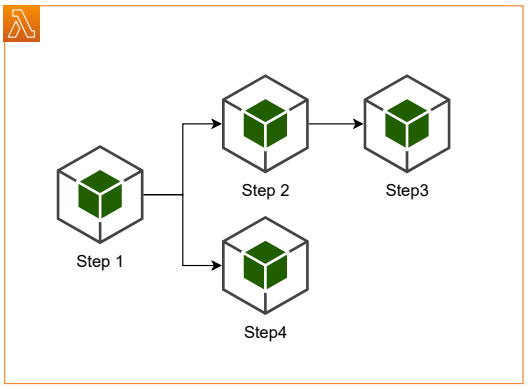
[35] Resolve lambda throttling "rate exceeded" and 429 errors? https://aws.amazon.com/premiumsupport/knowledge-center/lambda-troubleshoot-throttling. Accessed: 13-Aug-2021.

[36] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009.

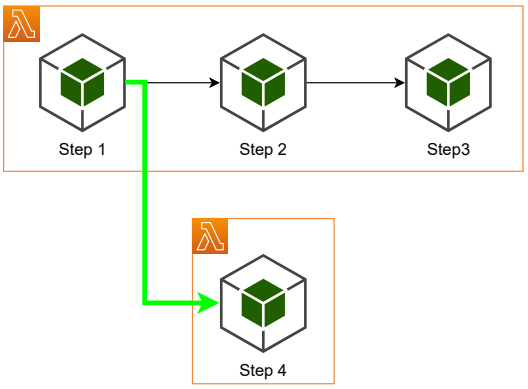
A

Appendix 1 - Partitions of Parallel Application

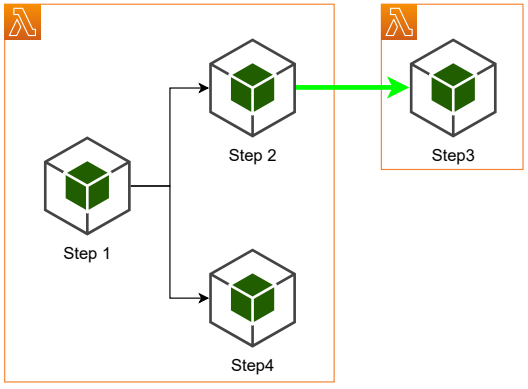
#P01: [S1, S2, S3, S4]



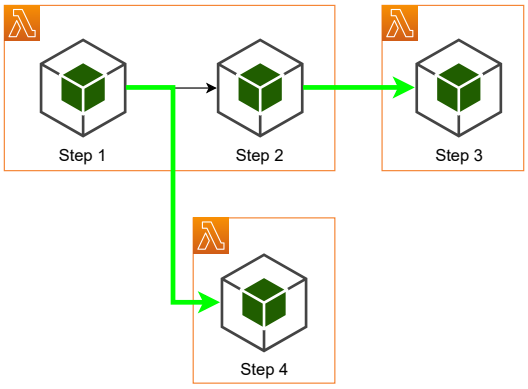
#P02: [S1, S2, S3] - [S4]



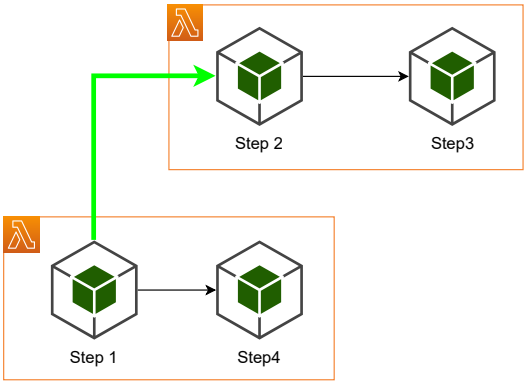
#P03: [S1, S2, S4] - [S3]



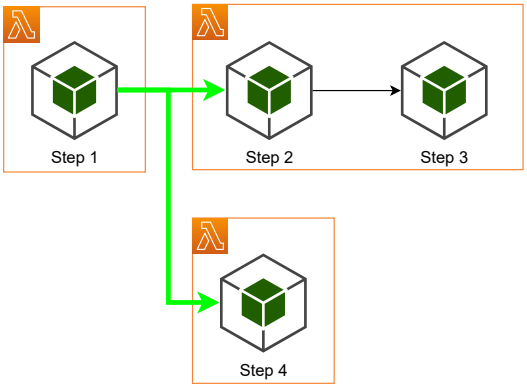
#P04: [S1, S2] - [S3] - [S4]



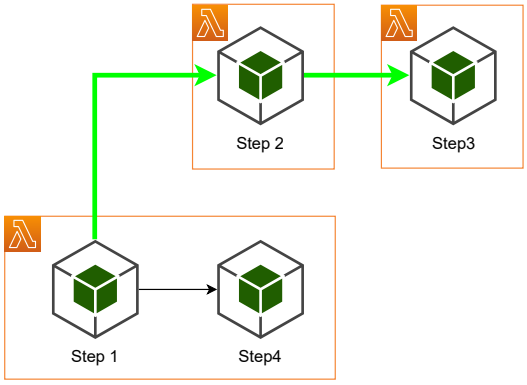
#P05: [S1, S4] - [S2] - [S3]



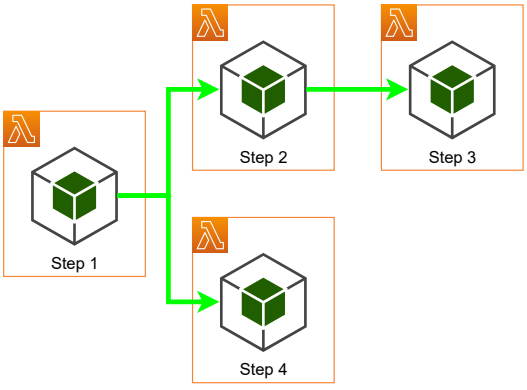
#P06: [S1] - [S2, S3] - [S4]



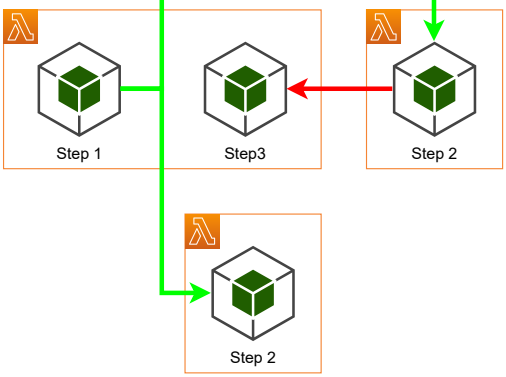
#P07: [S1, S4] - [S2] - [S3]



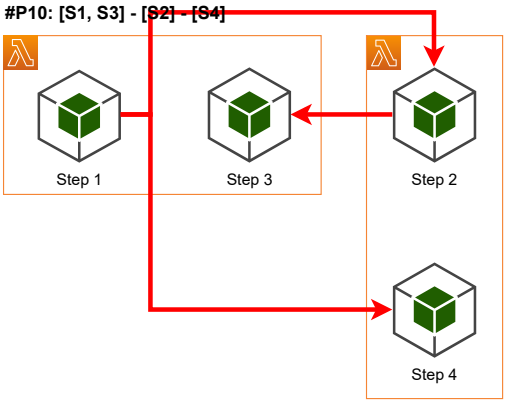
#P08: [S1] -[S2] - [S3] - [S4]



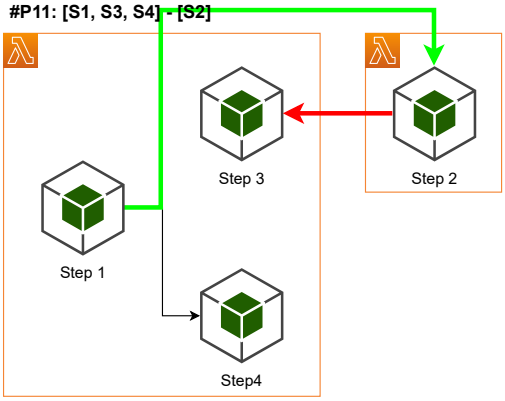
#P09: [S1, S3] - [S2] - [S4]



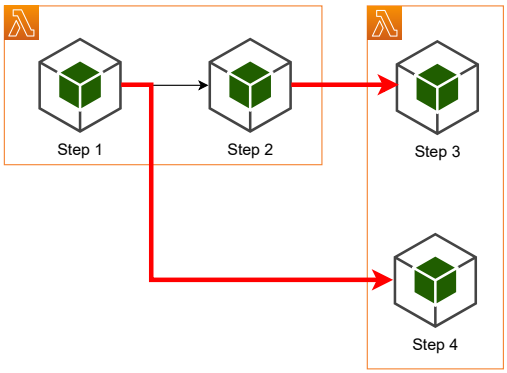
#P10: [S1, S3] - [S2] - [S4]



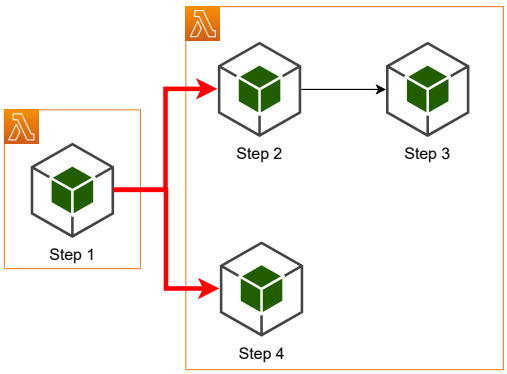
#P11: [S1, S3, S4] - [S2]



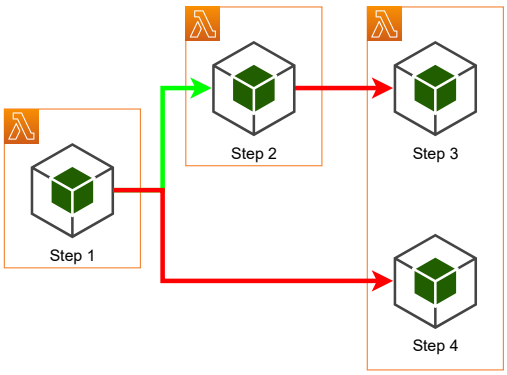
#P12: [S1, S2] - [S3, S4]



#P13: [S1] - [S2, S3, S4]



#P14: [S1] - [S2] - [S3, S4]



#P15: [S1] - [S2, S4] - [S3]